Otto Mayer

Programmieren in COMMON LISP

B·I

Hochschultaschenbücher
Band 638

B.I.-Hochschultaschenbücher
Band 638

Programmieren in COMMON LISP

von
Prof. Dr. Otto Mayer
Universität Kaiserslautern

B·I
Wissenschaftsverlag
Mannheim/Wien/Zürich

CIP-Titelaufnahme der Deutschen Bibliothek

Mayer, Otto:
Programmieren in COMMON LISP / von Otto Mayer. –
Mannheim; Wien; Zürich: BI-Wiss.-Verl., 1988
(BI-Hochschultaschenbücher; Bd. 638)
ISBN 3-411-00638-2
NE: GT

Der Begriff »B.I.-Hochschultaschenbuch«
ist für den Verlag Bibliographisches Institut & F.A. Brockhaus AG
als Warenzeichen geschützt

Autor und Verlag übernehmen für die Fehlerfreiheit der
Programme keine Gewährleistung oder Haftung.
Der Verlag übernimmt keine Gewähr dafür, daß die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter
sind.

Alle Rechte, auch die der Übersetzung in fremde Sprachen,
vorbehalten. Kein Teil dieses Werkes darf ohne schriftliche
Einwilligung des Verlages in irgendeiner Form (Fotokopie,
Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke
der Unterrichtsgestaltung, reproduziert oder unter Verwendung
elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet
werden.
© Bibliographisches Institut & F.A. Brockhaus AG, Zürich 1988
Druck: Druckerei Krembel, Speyer
Bindearbeit: Pilger-Druckerei GmbH, Speyer
Printed in Germany
ISBN 3-411-00638-2

Gewidmet

meinen Kindern

Luitgard,

Teresa

und Christoph

Vorwort

Die Geschichte der Höheren Programmiersprachen reicht gut 30 Jahre zurück. Von den vielen in dieser Zeit entwickelten Sprachen konnten nur wenige Bedeutung erlangen. Auch von diesen wurden die meisten nach kurzer Blütezeit von neuen Sprachen und neueren Konzepten verdrängt. Nur FORTRAN und LISP, die beiden ältesten konnten rund drei Jahrzehnte mit praktisch unverändertem Konzept überleben.

Während FORTRAN letztlich aufgrund seiner Einfachheit seine Stellung bei technisch-numerischen Berechnungen behauptete, gewann LISP vor allem aufgrund seines besonderen mathematischen Konzepts (es beruht auf dem λ-Kalkül von Church) bei fortgeschrittenen Anwendungen der Informatik stetig an Bedeutung.

LISP gilt heute als Standardsprache der "Künstlichen Intelligenz" schlechthin. Dabei verwenden wir dieses Schlagwort lediglich als Sammelbegriff für komplexe Anwendungen der symbolischen Informationsverarbeitung. Dazu gehören insbesondere Gebiete wie

Wissensverarbeitung (Expertensysteme)
Bildverarbeitung
Robotertechnologie
Verarbeitung natürlicher Sprachen
Automatisches Beweisen
Symbolische Algebra
und andere.

Mit diesen Gebieten hat auch die Sprache LISP in den letzten Jahren stark an Bedeutung gewonnen und ist heute fester Bestandteil der Informatikstudiengänge geworden.

Dieses Buch entstand aus Vorlesungen über "Programmieren in LISP", die ich in den letzten Jahren an der Universität Kaiserslautern gehalten habe.
Es setzt keine besonderen Vorkenntnisse voraus und kann von Studenten der Informatik oder Mathematik ab dem 1. oder 2. Semester auch zum Selbststudium verwendet werden.

Dieses Buch ist auch Grundlage für zwei COSTOC-Kurse geworden. COSTOC steht für Computer Supported Teaching of Computer Science und bedeutet, daß der Computer selbst zur Vermittlung des Lehrstoffs, hier die Einführung in das Programmieren in LISP, verwendet wird.
Das Buch eignet sich deshalb auch hervorragend als Begleitmaterial zu diesen Kursen.

Bedanken möchte ich mich bei Fräulein Dipl.-Math. Claudia Penning sowie den Herren Dipl.-Math. Klaus Spies und Dipl.-Inform. Dr. Bernd Schend für die Durchsicht des Manuskripts und wertvolle Hinweise und bei Frau Rita Kohl für die große Sorgfalt und Geduld bei der mühevollen Erstellung des Manuskripts.

Kaiserslautern, im Juli 1988 Otto Mayer

INHALTSVERZEICHNIS

KAPITEL	0	EINLEITUNG	13
	0.1	Entwicklung und Bedeutung von LISP	13
	0.2	Ziele und Voraussetzungen	14
	0.3	Aufbau und Inhalt	14

TEIL I Grundlagen 19

KAPITEL	1	LISTEN FÜR DATEN UND PROGRAMME	21
	1.1	Erster Versuch am LISP-System	21
	1.2	Atome und Listen	24
	1.3	Zur Darstellung von Listen im Speicher	25
	1.4	Zusammenfassung	29
KAPITEL	2	WERTZUWEISUNG UND WERTERMITTLUNG	31
	2.1	Wertzuweisung und Seiteneffekt	31
	2.2	Quotieren als Schutz gegen Auswertung	35
	2.3	Zur Auswertung von LISP-Programmen	36
	2.4	Der Interpretierer als Systemfunktion	38
	2.5	Übersetzung von LISP	41
	2.6	Zusammenfassung	42
KAPITEL	3	AUFBAU UND ZERLEGUNG VON LISTEN	43
	3.1	Aufbau von Listen mit CONS	43
	3.2	Zerlegung von Listen mit CAR und CDR	45
	3.3	Mehr zu CONS, CAR und CDR	48
	3.4	Zusammenbau von Listen mit APPEND und LIST	51
	3.5	LIST*, eine Modifikation von LIST	53
	3.6	Garbage-Collection	54
	3.7	Zusammenfassung	55

KAPITEL	4	PRÄDIKATE	57
	4.1	Drei grundlegende Prädikate: ATOM, CONSP und LISTP	57
	4.2	Übereinstimmung von LISP-Strukturen	61
	4.3	Vergleich von Zahlen	63
	4.4	Mehr zum Testen auf Übereinstimmung	66
	4.5	Mitgliedschaft in einer Liste, die Funktion MEMBER	67
	4.6	Zusammenfassung	69
KAPITEL	5	BEDINGUNGEN	71
	5.1	Noch einige Prädikate	71
	5.2	Typprüfungen	74
	5.3	Die logischen Verknüpfungen AND, OR und NOT	75
	5.4	Bedingungen, die spezielle Form COND	78
	5.5	Weitere Funktionen zur Manipulation von Listen	80
	5.6	Zusammenfassung	82
KAPITEL	6	FUNKTIONEN	83
	6.1	Definition benannter Funktionen	83
	6.2	Definition unbenannter Funktionen, Lambda-Ausdrücke	85
	6.3	Beispiele zur Definition von Funktionen	86
	6.4	Argumentübergabe beim Funktionsaufruf	89
	6.5	Freie und gebundene Variable	90
	6.6	Zusammenfassung	93
KAPITEL	7	REKURSION	95
	7.1	Grundprinzipien zur Definition rekursiver Funktionen	95
	7.2	Einfache Beispiele für rekursive Funktionen	97
	7.3	Die Türme von Hanoi	100
	7.4	Zusammenfassung	104
KAPITEL	8	ARGUMENTE UND VARIABLE	107
	8.1	Bindung und Umgebung	107
	8.2	Globale Variable	110
	8.3	Einrichtung neuer Variablenbindungen mit LET-Ausdrücken	113
	8.4	Beispiele zur Skopierung	115
	8.5	Funktionale Argumente	118
	8.6	Funktionale Objekte	121
	8.7	Zusammenfassung	126

KAPITEL	9	KONTROLLSTRUKTUREN	127
	9.1	PROG-Formen: PROG, PROG1 und PROGN	128
	9.2	Map-Funktionen	129
	9.3	Allgemeine Iteration	131
	9.4	Rekursion versus Iteration: ein Beispiel	133
	9.5	Zusammenfassung	135
KAPITEL	10	GRUNDLAGEN DER EIN- UND AUSGABE	137
	10.1	Ströme als Kommunikationswege	137
	10.2	Einlesen mit READ	139
	10.3	Ausgabe mit PRINT	142
	10.4	Exotische Symbolbezeichnungen	145
	10.5	Einfache Variationen	146
	10.6	Zusammenfassung	148
TEIL II		WEITERER AUSBAU	149
KAPITEL	11	LISTEN	151
	11.1	Listen und Bäume	151
	11.2	Mengen	153
	11.3	Assoziationslisten	154
	11.4	Kopieren von Listen	157
	11.5	Zusammenfassung	163
KAPITEL	12	DESTRUKTIVE FUNKTIONEN	165
	12.1	RPLACA und RPLACD ändern CAR und CDR	165
	12.2	Zirkuläre Listen	169
	12.3	Wie arbeitet eigentlich APPEND ?	170
	12.4	NCONC, die destruktive Variante von APPEND	172
	12.5	MAPCAN und MAPCON, zwei destruktive Map-Funktionen	175
	12.6	NREVERSE zur Umkehrung von Listen	178
	12.7	Zusammenfassung	179

KAPITEL	**13**	**FORMATIERTE AUSGABE**	**181**
	13.1	Mehr zur PRINT, PRIN1 und PRINC	181
	13.2	Schönheit überzeugt leichter !	184
	13.3	Die Funktion FORMAT	184
	13.4	Weitere Steueranweisungen für FORMAT	188
	13.5	Fehlermeldungen	193
	13.6	Zusammenfassung	196
KAPITEL	**14**	**PARAMETERSPEZIFIKATIONEN BEIM FUNKTIONS-KONZEPT**	**197**
	14.1	Optionale Argumente	197
	14.2	Beliebig viele Argumente	200
	14.3	Schlüsselwort-Argumente, ein anderes Konzept für optionale Argumente	202
	14.4	Zusammenspiel verschiedener Parameterspezifikationen	206
	14.5	Zusammenfassung	207
KAPITEL	**15**	**MAKROS**	**209**
	15.1	Zur Definition von Makros	209
	15.2	Das Makro IF	210
	15.3	Makros für spezielle Formen	214
	15.4	Backquote	216
	15.5	Seiteneffekte bei Makros	218
	15.6	Zusammenfassung	219
KAPITEL	**16**	**SYMBOLE**	**221**
	16.1	Namen für Symbole	221
	16.2	Zugriff auf Symbole über Packages	223
	16.3	Zur Erzeugung von Symbolen	225
	16.4	Zur Struktur von Symbolen	225
	16.5	Die Property-Liste	227
	16.6	Symbole für Werte und Funktionsbeschreibungen	230
	16.7	Zusammenfassung	232

KAPITEL	**17**	**EIN LISP-INTERPRETER**	**235**
	17.1	Ein Interpreter für LISP in LISP	235
	17.2	Die Funktion O-EVAL beschreibt den Interpreter	236
	17.2.1	Zur Auswertung von Symbolen	237
	17.2.2	Zur Auswertung von nichtleeren Listen	238
	17.2.2.1	Spezielle Formen und ihre Auswertung	239
	17.2.2.2	Primitive Operationen und ihre Auswertung	240
	17.2.2.3	Lambda-Ausdrücke und ihre Auswertung	242
	17.2.2.4	Frei definierte Funktionen und ihre Auswertung mit O-APPLY	243
	17.2.2.5	Vollständige Auswertung von nichtleeren Listen	245
	17.3	Zusammenfassung	245
KAPITEL	**18**	**ARRAYS**	**247**
	18.1	Einrichtung von Arrays	247
	18.2	Zugriff auf Array-Komponenten, das allgemeine Zuweisungsmakro SETF	250
	18.3	Vektoren und Sequenzen	253
	18.4	Strings	255
	18.5	Vergleich von Strings	256
	18.6	Informationen über ARRAYS	258
	18.7	Zusammenfassung	259
KAPITEL	**19**	**WEITERE KONTROLLSTRUKTUREN**	**261**
	19.1	Ausgänge	261
	19.1.1	Statische Ausgänge aus Blöcken	261
	19.1.2	Dynamische Ausgänge	263
	19.1.3	UNWIND-PROTECT zur Sicherung von Seiteneffekten bei THROWS	264
	19.2	Einfache Wiederholungen	266
	19.3	Iteration und Endrekursion	268
	19.4	Zusammenfassung	271

KAPITEL	20	MUSTERVERGLEICH (Pattern-Matching)	273
	20.1	Einfacher Mustervergleich	274
	20.2	Patternvariable	276
	20.3	Eine Funktion MATCH	277
	20.4	Anwendung des Mustervergleichs	279
	20.5	Ein Dialog	283
	20.6	Zusammenfassung	284
KAPITEL	21	DATENTYPEN UND DATENSTRUKTUREN	285
	21.1	Datentypen in COMMON LISP	285
	21.1.1	Die wichtigsten Datentypen	286
	21.1.2	Typbezeichner	288
	21.2	Strukturierte Objekte	289
	21.3	Mehr zum DEFSTRUCT-Konzept	292
	21.4	Vielfach-Werte (multiple values)	295
	21.5	Zur Übernahme von Vielfach-Werten	297
	21.6	Zusammenfassung	300
KAPITEL	22	ZUR ENTWICKLUNG VON PROGRAMMEN	303
	22.1	Strukturiertes Programmieren	303
	22.2	Unterstützung durch Programmentwicklungssysteme	305
	22.3	Die Komponenten eines Programmentwicklungssystems	307
	22.4	Das TRACE-Paket	316
	22.5	Das STEP-Paket	318
	22.6	Zusammenfassung	320
LITERATURHINWEISE			323
STICHWORTVERZEICHNIS			325

KAPITEL 0:

Einleitung

0.1 Entwicklung und Bedeutung von LISP

LISP ist eine der ältesten Programmiersprachen. Sie wurde in ihren Anfängen am Massachusetts Institute of Technology von John McCarthy entwickelt. Die Vorarbeiten begannen ab 1956. Konzeption, Implementierung und erste Anwendungen fallen in die Jahre 1958 - 1962.
Das Ziel war damals schon, eine für Probleme der "Künstlichen Intelligenz" geeignete Sprache zu schaffen. Deshalb wird in LISP auch nicht mit numerischen Werten, sondern mit symbolischen Ausdrücken gerechnet, die ihrerseits durch Listenstrukturen dargestellt werden. Darauf weist auch der Name LISP als Akronym für List-Processing hin.
Unter den Sprachen, die mit symbolischen Ausdrücken rechnen, oder Symbolmanipulationen durchführen, ist LISP weitaus am bedeutendsten. Dies beruht wesentlich auf folgenden Grundeigenschaften:

- LISP ermöglicht grundsätzlich interaktive Arbeitsweise: Der Benutzer gibt einen Ausdruck ein, und das LISP-System interpretiert ihn (wertet ihn aus) und gibt das Resultat aus. So können größere Programme Schritt für Schritt erstellt und getestet werden.

- LISP-Programme und LISP-Daten haben dieselbe Form. Ein LISP-Programm kann deshalb ein anderes LISP-Programm als Daten verwenden.

Heute gilt LISP als die Standardsprache der Künstlichen Intelligenz schlechthin. Das bedeutet: Programme, die ein Verhalten zeigen, das geheimhin als "intelligent" angesehen wird, werden fast ausschließlich in LISP formuliert. Mangels einer verbindlichen Definition ist LISP keine völlig einheitliche Sprache geblieben wie FORTRAN, ALGOL oder PASCAL. Aus unterschiedlichen Gründen kam es schon frühzeitig zu Weiterentwicklungen in verschiedenen Richtungen: Es entstanden verschiedene Dialekte der Sprache LISP, zugeschnitten auf spezielle Bedürfnisse bzw. Wünsche oder Rechnereigenschaften. Ein 1981 gestarteter Versuch zur Vereinheitlichung und Fortschreibung eines Sprachkerns, der allen LISP-Dialekten gemeinsam (englisch common) sein sollte, führte schließlich zu COMMON LISP.

Dieses Buch gibt auf der Grundlage von COMMON LISP eine Einführung in das Programmieren in LISP.

0.2 Ziele und Voraussetzungen

Die Sprache COMMON LISP kann und soll in diesem Buch nicht vollständig vorgestellt werden. Dazu ist sie einfach zu umfangreich. Das COMMON LISP Manual (siehe Literaturhinweise) benötigt rund 450 Druckseiten für eine vollständige Definition, ist aber zum Erlernen der Sprache völlig ungeeignet.

Dieses Buch vermittelt solide Kenntnisse der Sprache LISP und versetzt den aufmerksamen Leser in die Lage, anspruchsvolle Programme zu entwerfen, zu verstehen und darüberhinaus auftretende Fragen selbständig im COMMON LISP Manual oder in speziellen Systembeschreibungen zu klären.

LISP unterscheidet sich ganz wesentlich von allen anderen bekannten Programmiersprachen. Deshalb kann die Kenntnis anderer Programmiersprachen das Erlernen von LISP nicht entscheidend erleichtern, sie kann sogar anfangs dem richtigen Verständnis von LISP im Wege stehen.

Diese Einführung setzt keine speziellen Kenntnisse über Rechner und ihre Programmierung voraus. Auf jeden Fall sollte ein Informatik-Student im zweiten Semester den Inhalt erfolgreich verarbeiten können. Wir meinen sogar, daß man LISP anhand dieses Buches auch als erste Programmiersprache erlernen kann.

0.3 Aufbau und Inhalt

Diese Einführung in LISP besteht aus zwei Teilen.

Teil I mit 10 Kapiteln vermittelt die Grundlagen der Sprache und bildet auch für sich allein eine abgerundete Einführung in die wesentlichen Konzepte der Symbolverarbeitung mit LISP.

Kapitel 1 vermittelt zunächst eine intuitive Vorstellung von der Funktionsweise und Bedienung eines LISP-Systems und behandelt dann Listen als grundlegende Datenstruktur zur Darstellung von Programmen und Daten.

Kapitel 2 beschäftigt sich mit der Zuordnung von Werten an die Bestandteile von Listen und der Auswertung von Listen durch den Interpretierer.

Kapitel 3 behandelt Operationen für Aufbau und Zerlegung von Listen.

Kapitel 4 beschäftigt sich mit grundlegenden Prädikaten für einfache Typprüfungen und zum Vergleich von Datenstrukturen bzw. Zahlen.

Kapitel 5 zeigt, wie die Auswertung von Funktionen oder Teilen eines Programms von Bedingungen abhängig gemacht werden kann.

Kapitel 6 behandelt die Definition von Funktionen durch den Benutzer.

Kapitel 7 ist der Rekursion gewidmet, einem der wichtigsten Konzepte für das Programmieren in LISP.

Kapitel 8 behandelt die Zuordnung von Argumenten und Variablen in sogenannten Umgebungen und untersucht die Verwendung von Funktionen als Datenobjekte.

Kapitel 9 führt verschiedene Kontrollstrukturen, insbesondere zur Iteration, ein.

Kapitel 10 vermittelt die Grundlagen der Ein- und Ausgabe.

Teil II mit 12 Kapiteln erweitert die Grundlagen um eine Reihe von Konzepten, welche den praktischen Einsatz der Sprache beträchtlich erleichtern und zeigt einige charakteristische Anwendungen.

Kapitel 11 dient dazu, unser Wissen über Listen zu vertiefen. Wir erkennen, daß die Listen der Sprache LISP externe Repräsentationen von binären Bäumen sind; sie bilden die grundlegende interne Speicherstruktur von LISP.

Kapitel 12 behandelt destruktive Funktionen; darunter verstehen wir Funktionen, die vorhandene Speicherstrukturen durch Umsetzen von Pointern "destruktiv" verändern.

Kapitel 13 zeigt die vielfältigen Möglichkeiten der Formatierung von Ausgaben.

Kapitel 14 lehrt, wie man auch für selbstdefinierte Funktionen optionale Argumente sowie eine beliebige Anzahl von Argumenten spezifizieren kann.

Kapitel 15 behandelt Makros; hier lernen wir, LISP-Programme zu schreiben, die LISP-Formen in andere LISP-Formen überführen, welche dann anstelle der ursprünglichen Formen ausgewertet werden.

Kapitel 16 beschäftigt sich mit Symbolen, ihren Namen, ihrer Organisation in sogenannten Packages und ihren Property-Listen sowie den Möglichkeiten der Wert- bzw. Funktionszuordnung.

Kapitel 17 zeigt als typische Anwendung von LISP die Konstruktion eines LISP-Interpretierers für eine einfache Teilmenge von LISP. Dieses Beispiel der Verarbeitung von LISP-Programmen zeigt die faszinierenden Möglichkeiten von LISP und ist Modell für die Konstruktion von Interpretierern für verschiedene LISP-Dialekte bzw. andere Programmiersprachen.

Kapitel 18 behandelt den Datentyp Array mit verschiedenen Untertypen, insbesondere Vektoren und Strings, sowie verschiedene Anwendungen.

Kapitel 19 beschäftigt sich mit höheren Kontrollstrukturen, insbesondere mit statischen und dynamischen Ausgängen aus sogenannten Blöcken, und dem Zusammenhang zwischen Iteration und Endrekursion.

Kapitel 20 zeigt Anwendungen von LISP beim Mustervergleich und einem einfachen Beispiel zur Verarbeitung von natürlicher Sprache.

Kapitel 21 gibt einen Überblick über Datentypen und führt das sogenannte DEFSTRUCT-Konzept zur Definition beliebig strukturierter Objekte ein.

Kapitel 22 gibt einen Überblick über Aufbau und Verwendung von Programmiersystemen.

Der Lehrstoff in Teil I kann sinnvollerweise nur in der gegebenen Reihenfolge durchgearbeitet werden, da die einzelnen Kapitel streng aufeinander aufbauen. Die jeweils eingeführten Sprachkonstrukte werden in den nachfolgenden Kapiteln konsequent in Beispielen und zum Aufbau komplizierterer Sprachelemente eingesetzt.

Der Lehrstoff in Teil II kann in gewissen Grenzen auch anders angeordnet werden. Insbesondere könnte man Kapitel 13 (Formatierte Ausgabe) sofort im Anschluß an Kapitel 10 (Grundlagen der

Ein- und Ausgabe) und wesentliche Teile von Kapitel 22 (Zur Entwicklung von Programmen) unmittelbar im Anschluß an den Stoff von Kapitel 13 behandeln.
Die Kapitel 17 und 20 zeigen typische Anwendungen von LISP und werden nicht als Grundlage für nachfolgende Kapitel benötigt. Auf sie könnte später eingegangen werden.

Teil I

Grundlagen

KAPITEL 1:

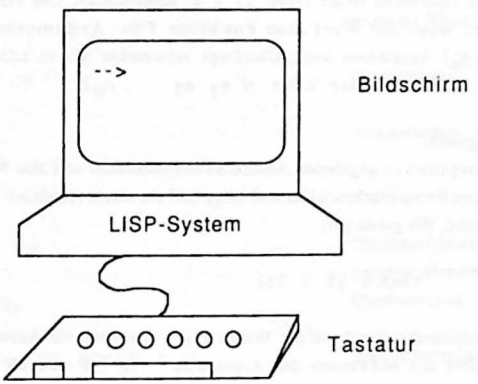
Listen für Daten und Programme

Dieses Kapitel vermittelt eine erste Vorstellung von der Arbeitsweise eines LISP-Systems und behandelt dann die Bausteine von LISP-Programmen und LISP-Daten, nämlich Atome und Listen, in ihrer externen und internen Darstellung.

1.1 Erster Versuch am LISP-System

Kern eines jeden LISP-Systems ist ein **Interpretierer** (auch **Interpreter** genannt), der eingegebene Ausdrücke evaluiert (d.h. auswertet) und die Resultatwerte ausgibt.
Ein **LISP-Programm** ist eine Folge von Ausdrücken, die dem LISP-System zur Auswertung eingegeben werden. Die Ausführung eines LISP-Programms besteht darin, daß nacheinander die einzelnen Ausdrücke ausgewertet werden. Diese Ausdrücke sind entweder Atome, die einen **Wert** haben, oder Listen. In Listen, die auszuwerten sind, bezeichnet das erste Listenelement jeweils eine Funktion, die restlichen Listenelemente geben die Argumente an, für welche diese Funktion auszuwerten ist. **Wert** der Liste ist der Wert der Funktion für diese Argumente.

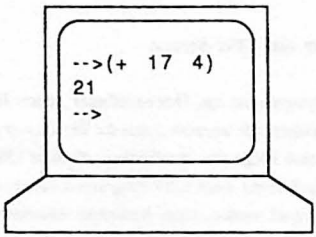
Angenommen vor uns steht ein LISP-System mit Tastatur und Bildschirm für Eingabe bzw. Ausgabe.



Wenn das **Bereitschaftszeichen** --> (engl. "Prompt")
auf dem Bildschirm erscheint, kann ein Ausdruck zum Auswerten eingegeben werden. Über die Tastatur geben wir dazu die Zeichenfolge

(+ 17 4)

ein und schließen diese Eingabe mit einem Druck auf die "Return-Taste" ab. Das System wertet dann sofort die Addition von 17 und 4 aus und gibt in der neuen Zeile das Resultat 21 aus. Anschließend erscheint ein neues Bereitschaftszeichen.

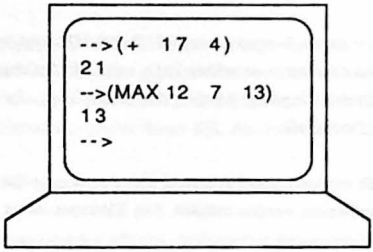


Das erste Listenelement im Ausdruck (+ 17 4) bezeichnet die Addition als diejenige Funktion, welche auf die durch die beiden restlichen Listenelemente gegebenen Argumente 17 und 4 anzuwenden ist.
Diese Angabe der Funktion vor den Argumenten - man spricht von **Präfix-Notation** - ist im Falle der zweistelligen arithmetischen Operatoren ungewohnt; hier ist die **Infix-Notation**, die Angabe zwischen den beiden Operanden in der Form 17 + 4 gebräuchlich. Die Präfix-Notation ist uns jedoch vertraut, wenn der **Wert einer Funktion f für Argumente** $a_1, \ldots, a_n$ durch $f(a_1, \ldots, a_n)$ bezeichnet wird; allerdings verwenden wir in LISP hierfür eine **Präfix-Notation in Gestalt der Liste** $(f\ a_1\ a_2\ \ldots\ a_n)$.

Die nächsten Eingaben:
Sobald die Auswertung eines eingegebenen Ausdrucks abgeschlossen und sein Wert ausgegeben ist, erscheint ein neues Bereitschaftszeichen und zeigt, daß ein neuer Ausdruck zum Auswerten eingegeben werden kann. Wir geben jetzt

(MAX 12 7 13)

ein. Durch den abschließenden Druck auf die Return-Taste setzen wir die Auswertung in Gang. Daraufhin wird sofort das Maximum der Argumente 12 , 7 und 13 bestimmt und ausgegeben. Anschließend erscheint wieder ein neues Bereitschaftszeichen.



Weitere Beispiele zur Auswertung von arithmetischen Funktionen.
Angegeben sind jeweils - das Bereitschaftszeichen des Systems
- der dazu eingegebene Ausdruck
- ein Kommentar
- der ausgegebene Wert

```
-->(- 11 9)                              ; Subtraktion
2
-->(* 11 9)                              ; Multiplikation
99
-->(/ 7.5 3)                             ; Division
2.5
-->(+ 7.3)                               ; positives Vorzeichen
7.3
-->(- 4711)                              ; negatives Vorzeichen
-4711
-->(EXPT 2 3)                            ; Exponentiation
8
-->(ABS -5)                              ; Absolutbetrag
5
-->(ABS 8)
8
-->(GCD 91 49)                           ; "Greatest Common Divisor"
7                                        ; (größter gemeinsamer Teiler)
-->(SQRT 9.0)                            ; Quadratwurzel
3.0
-->(+ (+ 17.0 4) (EXPT 2 3))             ; geschachtelte Liste
29.0
```

In den obigen Beispielen treten Kommentare auf. Das LISP-System ignoriert Strichpunkte (Semicolons) und alles was dahinter in derselben Zeile steht. Mit Strichpunkt und nachfolgendem Text kann also ein Kommentar eingefügt werden; das ist ein Text, der das Programm erläutert, aber die Auswertung nicht beeinflußt.

Das letzte Beispiel enthält eingeschachtelte Listen; die Argumente für die Addition sind selbst Listen, die ihrerseits ausgewertet werden müssen. Die Elemente einer Liste können also selbst wieder Listen sein.

1.2 Atome und Listen

Da Listen in LISP-Programmen eine zentrale Rolle spielen, sollen ihr Aufbau und ihre Struktur näher betrachtet werden.

Die kleinsten nicht mehr weiter zerlegbaren Bausteine von Listen heißen **Atome**.

Als Atome treten auf

- *Zahlen* auch Numerische Atome genannt
  u.a. als Ganze Zahlen und als Gleitpunktzahlen

- *Symbole* auch Symbolische Atome genannt
  Hier sind praktisch alle Zeichenfolgen zugelassen, die von Zahlen, Satzzeichen und Klammern unterscheidbar sind und nicht mit Anführungszeichen eingerahmt sind.
  Wir verwenden vor allem Folgen von Buchstaben, Ziffern und Gedankenstrichen, wobei das erste Symbol meistens ein Buchstabe ist und Gedankenstriche nicht unmittelbar nebeneinander auftreten; Beispiele sind
  T, NIL, EULE, ARGUMENT-1, X, S, X17, X-17 usw.

  Einige Symbole wie +, -, *, /, <, >, = werden, wenn sie isoliert auftreten, nur zur Darstellung von Operatoren benutzt.

- *Strings* das sind Zeichenfolgen, die mit hochgesetzten Anführungszeichen eingerahmt sind.
  Beispiele sind "abcde", "653", "~%~Sa13", "~A" u.a.

Eine **Liste** besteht aus einer öffnenden Klammer gefolgt von keinem, einem oder mehreren Listenelementen, die durch Leerzeichen getrennt sind, und einer schließenden Klammer. Ein Listenelement ist ein Atom oder wieder eine Liste. Die Liste mit null Elementen, wir stellen sie durch ( ) oder gleichbedeutend durch das Atom NIL dar, heißt **leere Liste**.

Atome und Listen bezeichnet man zusammen als **Ausdrücke**.

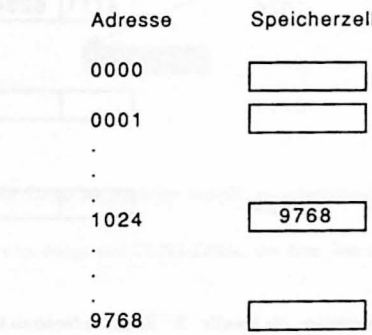
Damit Ausdrücke ausgewertet werden können, müssen ihre Bestandteile also zunächst vor allem die Atome einen Wert haben. Numerische Atome, also Zahlen wie 17 oder 4.13 haben stets einen Wert, nämlich den durch sie dargestellten Zahlenwert. Auch ein String hat stets sich selbst als Wert. Symbole wie EULE und ZWEI haben einen Wert, wenn ihnen ein solcher zugeordnet ist; darauf kommen wir noch zurück.

Ein Ausdruck, der ausgewertet werden kann, wird auch **Form** genannt. Ist die Form eine Liste, so bestimmt das erste Listenelement die Funktion, die zur Auswertung heranzuziehen ist.

1.3 Zur Darstellung von Listen im Speicher

Im folgenden behandeln wir die Darstellung von Atomen und Listen im Speicher eines LISP-Systems.

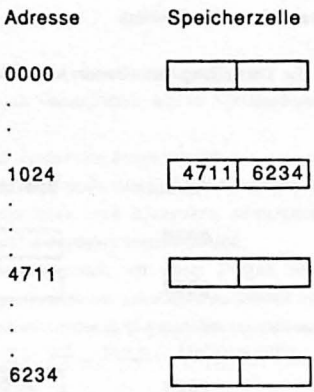
Der Speicher eines Rechners besteht aus Speicherzellen. Diese sind durchnumeriert. Die Nummer einer Zelle wird auch Adresse genannt.



Die Inhalte von Speicherzellen sind Zahlen; diese können als Codierungen für irgendwelche Objekte, aber auch als Adressen von Speicherzellen gedeutet werden. Deutet man den Inhalt einer Speicherzelle als Adresse einer anderen Speicherzelle, so sagt man auch, die erste Speicherzelle enthält einen Zeiger (oder Pointer) auf die andere Speicherzelle. In diesem Sinne enthält die Speicherzelle 1024 einen Zeiger auf die Speicherzelle 9768.

Atome und Listen werden dargestellt durch eine oder mehrere Speicherzellen, die in geeigneter Weise durch Zeiger verbunden sind. Dazu gibt es verschiedene Möglichkeiten.
Wir können annehmen, daß jede Speicherzelle zwei Zeiger aufnehmen kann, einen linken und einen rechten Zeiger.

Dies ist keine Einschränkung:
Ob die beiden Zeiger tatsächlich in einem Speicherwort, in zwei aufeinanderfolgenden Speicherwörtern oder auf andere Weise untergebracht sind, davon abstrahieren wir, indem wir unter einer Speicherzelle von jetzt ab einen Speicherbereich verstehen, der zwei Zeiger aufnehmen kann.

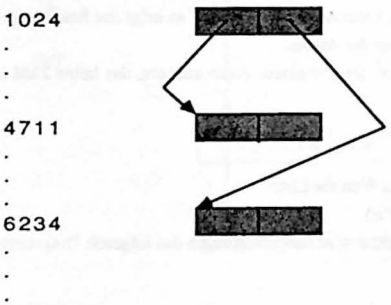


Derartige Speicherzellen, die jeweils 2 Zeiger aufnehmen können, spielen beim Aufbau der Speicherstrukturen für Listen eine besondere Rolle. Wir nennen sie auch CONS-Zellen, weil dieser

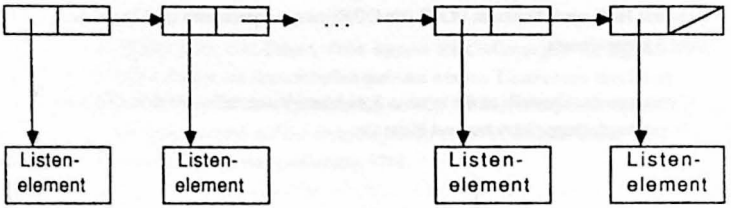
Aufbau mit Hilfe einer Funktion CONS (für CONStruct) vorgenommen wird. Diese werden wir in Kapitel 3 kennenlernen.

In Diagrammen zur Darstellung der internen Speicherstrukturen stellen wir diese CONS-Zellen und ihre Zeiger durch Doppelkästchen und Pfeile dar.

Eine Speicherbelegung mit Zeigern, wie oben angedeutet, wird also anschaulich wie folgt dargestellt:



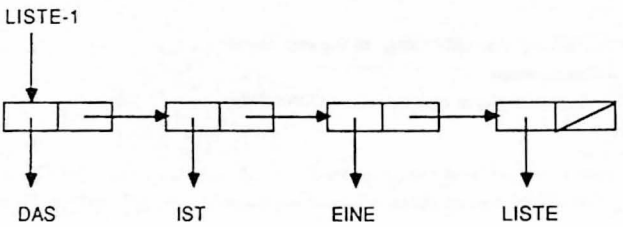
Was nun die Darstellung von LISP-Daten im Speicher betrifft, so interessieren uns nur die wesentlichen Zusammenhänge:
Eine Liste wird dargestellt durch eine Folge von CONS-Zellen, die über ihre rechten Zeiger verknüpft sind.



Jede der Speicherzellen in der Folge vertritt ein Listenelement. Dieses Listenelement selbst wird durch den linken Zeiger aus der Speicherzelle angezeigt.
Das Ende der Liste wird durch einen speziellen Eintrag für den letzten rechten Zeiger angezeigt. Im allgemeinen wird dazu die Zahl 0 eingetragen, die auch das Symbol NIL darstellt, mit dem wir die leere Liste bezeichnen. Im Diagramm zur Darstellung der internen Struktur verwenden wir nur einen Schrägstrich.

Ein Atom wird in der Darstellung der internen Speicherstruktur einfach durch Niederschrift seines Namens dargestellt. Ist ein Listenelement ein Atom, so zeigt der linke Zeiger der entsprechenden Speicherzelle auf den Namen des Atoms.
Umgekehrt zeigt ein Zeiger, der von einem Atom ausgeht, das keine Zahl ist, stets auf den Wert dieses Atoms.

Beispiel:
Das Atom LISTE-1 habe als Wert die Liste
(DAS IST EINE LISTE)
Die zugehörige interne Struktur wird dargestellt durch das folgende Diagramm.

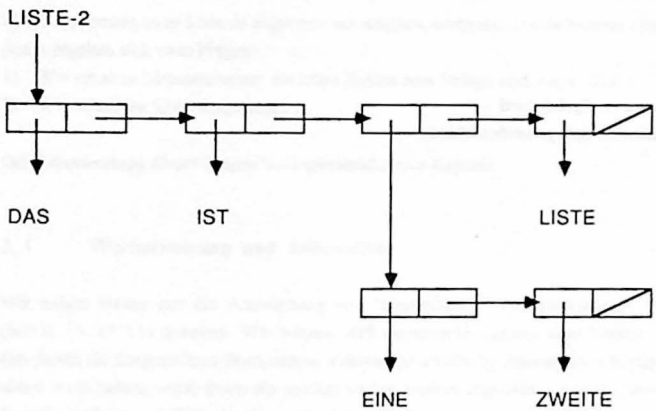


Die Elemente einer Liste sind wie wir wissen nicht immer Atome, sondern können selbst wieder Listen sein.
Im Diagramm zeigen dann die linken Zeiger aus den zugehörigen Speicherzellen eben nicht auf Atome, sondern auf andere Listen.

Beispiel:
Das Atom LISTE-2 habe als Wert die Liste

(DAS IST (EINE ZWEITE) LISTE)

Das folgende Diagramm zeigt die zugehörige interne Struktur.



1.4 Zusammenfassung

Grundbausteine der Sprache LISP sind Atome und Listen. Atome und Listen bilden zusammen die sogenannten symbolischen Ausdrücke, häufig S-Ausdrücke, hier kurz Ausdrücke genannt.
Ausdrücke, die ausgewertet werden können, werden auch Formen genannt. Dabei können die Begriffe Ausdruck und Form nebeneinander für dasselbe Objekt sinnvoll sein. Bei Verwendung des Begriffs Ausdruck für eine Liste bleibt offen, ob sie als Kollektion von Daten oder als Auswertungsvorschrift angesehen wird; die Bezeichnung Form weist deutlicher auf die Auswertbarkeit hin. Eine strenge Unterscheidung zwischen Daten und Auswertungsvorschriften

gibt es in LISP nicht, sie widerspricht sogar dem Charakter der Sprache LISP. Zu ihren besonderen Kennzeichen gehört gerade, daß LISP-Programme und LISP-Daten dieselbe Gestalt haben, so daß Programmstücke als Daten behandelt werden können.

Wir haben insbesondere gelernt,

- wie Atome und Listen definiert sind,
- wie Listen intern durch Zeigerstrukturen dargestellt werden,
- daß Ausdrücke bzw. Listen ausgewertet werden.

Wir werden uns in den nächsten Kapiteln zunächst mit der

- Zuordnung von Werten an Symbole sowie der
- Auswertung von Listen

beschäftigen und dann kennenlernen,

wie Listen

- aus Bausteinen aufgebaut und
- in ihre Bausteine zerlegt werden können.

KAPITEL 2:

Wertzuweisung und Wertermittlung

Wir haben in Kapitel 1 **einfache** Listen wie

```
(+ 17 4)
(MAX 13 17 11)
```

oder

```
(DA IST (NOCH EINE LISTE) EINGESCHACHTELT)
```

kennengelernt und zum Teil ausgewertet.

Eine Auswertung einer Liste ist allgemein nur möglich, wenn alle Listenelemente einen Wert haben. Somit ergeben sich zwei Fragen:

1) Wie erhalten Listenelemente, die keine Zahlen oder Strings sind, einen Wert ?
2) Wie wird eine Liste ausgewertet ?

Die Beantwortung dieser Fragen ist Gegenstand dieses Kapitels.

2.1 Wertzuweisung und Seiteneffekt

Wir haben bisher nur die Auswertung von "numerischen" Ausdrücken wie (+ 17 4) und (MAX 13 17 11) gesehen. Wir wissen, daß numerische Atome, also Zahlen, grundsätzlich den durch sie dargestellten Wert haben, während symbolische Atome (kurz Symbole), nur dann einen Wert haben, wenn ihnen ein solcher vorher explizit zugeordnet worden ist. Eine derartige Zuordnung kann mit Hilfe der Systemfunktion SETQ vorgenommen werden; wir sprechen dabei von einer **Wertzuweisung** (assignment).
Als **Systemfunktionen** bezeichnen wir übrigens Funktionen, die an jedem LISP-System standardmäßig zur Verfügung stehen.

Beispiel:
Bei Auswertung des Eingabe-Ausdrucks

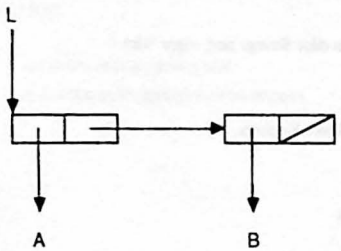
```
(SETQ L '(A B))
```

wird

- ein Wert für diesen Ausdruck ermittelt; dieser Wert ist definiert als Wert des zweiten Arguments '(A B). Das Anführungszeichen ' beim Argument '(A B) beeinflußt die Wertermittlung für dieses Argument. Wir gehen darauf im nächsten Abschnitt näher ein. Der Wert von '(A B) ist, wie noch begründet werden wird, gleich (A B).

- dem Symbol L als Wert der Wert (A B) des zweiten Arguments '(A B) zugewiesen.

Diese Zuweisung wird dadurch realisiert, daß im Speicher aus bisher freien Speicherzellen eine Zeigerstruktur, wie nachstehend skizziert, eingerichtet wird. Diese Struktur bleibt nach Beendigung der Auswertung der SETQ-Form bestehen.



Die Abarbeitung des folgenden Programmstücks zeigt nochmals deutlich die Wirkung von SETQ.

```
-->(SETQ L '(A B))
(A B)                    ; Wert der SETQ-Form

->L                      ; nach Eingabe des Symbols L wird sein Wert
(A B)                    ; ermittelt und ausgegeben.
```

Wichtiger Hinweis:

Wir begegnen hier zum erstenmal einer Funktion, die einen sogenannten Seiteneffekt hat.
Die bisher betrachteten Funktionen wie EXPT, SQRT, +, ABS und andere liefern jeweils einen Wert, ihre Auswertung hinterläßt aber keine Veränderungen im Speicher. Die Auswertung einer

SETQ-Form liefert zwar auch einen Wert, nämlich den Wert ihres zweiten Arguments; SETQ wird aber vor allem deshalb ausgewertet, um Zuordnungen im Speicher herzustellen oder zu ändern. Das aber ist ein sogenannter Seiteneffekt.

Allgemein versteht man unter einem **Seiteneffekt** einer Funktion jede Veränderung, die durch die Berechnung eines Funktionswertes hervorgerufen wird und nach Beendigung dieser Berechnung fortbesteht.

Nur mit Seiteneffekten können in LISP Zuordnungen zwischen Symbolen und Werten eingerichtet oder geändert werden.

Wir geben noch das **Aufrufmuster** für die Funktion SETQ an. In derartigen Aufrufmustern können einzelne Bestandteile - insbesondere Argumente - durch erläuternden Text in spitzen Klammern ersetzt werden. Die spitzen Klammern sind nicht Bestandteil des Sprachelements, sondern kennzeichnen die Erläuterung als solche.

Aufrufmuster von SETQ:

(SETQ <Symbol> <Ausdruck>)

Das bedeutet: SETQ hat zwei Argumente; das erste ist ein Symbol, das zweite ein Ausdruck.

Wert:
Wert des Ausdrucks.

Seiteneffekt:
Nach Auswertung von
(SETQ <Symbol> <Ausdruck>)
hat das Symbol als Wert den Wert des Ausdrucks.

Bei vielen Gelegenheiten sind mehrere Wertzuweisungen zusammen oder hintereinander auszuführen.

Beispiel:

```
-->(SETQ X (+ 3 2 1))
6

-->(SETQ Y (- 9 X))
3
```

Derartige Folgen von "SETQ-Auswertungen" können aber auch durch einen einzigen SETQ-Aufruf bewirkt werden. Dabei verwenden wir SETQ mit dem

Aufrufmuster:

(SETQ <Symbol-1> <Form-1>
<Symbol-2> <Form-2> ... <Symbol-n> <Form-n>)

Zur Auswertung

wird zuerst der Wert von Form-1 ermittelt und an Symbol-1 zugeordnet, anschließend der Wert von Form-2 ermittelt und an Symbol-2 zugeordnet, und so weiter; zuletzt wird der Wert von Form-n ermittelt und an Symbol-n zugeordnet. Offensichtlich muß die Anzahl der Argumente gerade sein. Die Reihenfolge der Paare "Symbol und Form" in der Niederschrift des SETQ-Aufrufs ist wegen der Seiteneffekte wesentlich.

Wert der SETQ-Form ist der Wert der letzten Form.

Seiteneffekt:

Die ermittelten Werte der Formen werden - wie beschrieben - den zugehörigen Symbolen zugeordnet.

Beispiel:

```
-->(SETQ X (+ 3 2 1) Y (- 9 X))
3                          ; Der Wert 6 der ersten Form wird an X
                           ; zugewiesen.
                           ; Zur Auswertung der zweiten Form wird der Wert
                           ; von X verwendet.
```

```
-->X
6

-->Y
3
```

2.2 Quotieren als Schutz gegen Auswertung

Wir haben noch die Bedeutung des Anführungszeichens ' für die Auswertung eines damit eingeleiteten Ausdrucks zu klären.

Ein Gebilde der Gestalt

'<Ausdruck>

ist eine gebräuchliche **Abkürzung** und zwar für

(QUOTE <Ausdruck>);

das aber ist ein Aufruf der Systemfunktion QUOTE mit einem Ausdruck als Argument.

Wert von (QUOTE <Ausdruck>)
ist der gegebene Ausdruck selbst und zwar unverändert.

Dies entspricht auch der Bedeutung von "to quote" = zitieren = unverändert wiedergeben.
Wir sprechen hier von "Quotieren"; es schützt sein Argument gegen Auswertung.

Beispiele:

```
-->(+ 17 4)
21

-->(QUOTE (+ 17 4))
(+ 17 4)

-->(SETQ S (+ 17 4))          ; Der Wert des zweiten Arguments wird ermittelt
21                            ; und als Wert der SETQ-Form ausgegeben
```

```
-->(SETQ A '(+ 17 4));          ; Der Wert des zweiten Arguments ist die Liste
(+ 17 4)                        ; (+ 17 4)

-->S                            ; Der Wert des Symbols S wird ermittelt und
21                              ; ausgegeben

-->A                            ; A hat als Wert die Liste (+ 17 4)
(+ 17 4)

-->'(A B 'C D)                  ; QUOTE wird in Ausgabe i.a. nicht abgekürzt
(A B (QUOTE C) D

-->' '(A B C D)                 ; Eine Auswertung "verbaucht" eine Quotierung
(QUOTE (A B C D))

-->(+ (QUOTE 17) (QUOTE 4))     ; Zahl haben sich selbst als Wert. Quotierung bei
21                              ; Zahlen ohne Einfluß !

-->(+ '7 '4)
11
```

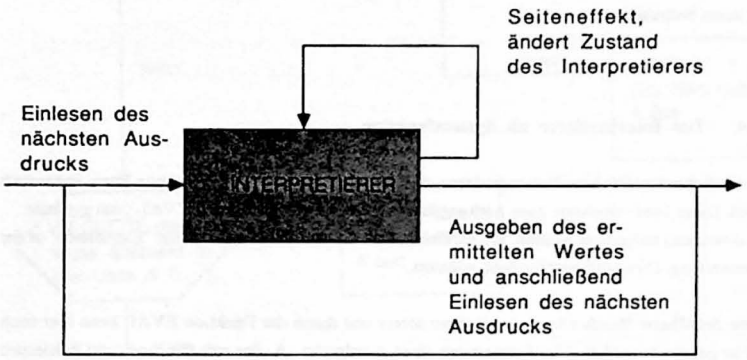
2.3 Zur Auswertung von LISP-Programmen

Nachdem wir inzwischen einige Erfahrung im Umgang mit Listen erworben haben, wollen wir jetzt noch etwas genauer erfahren, wie auswertbare Ausdrücke vom LISP-System ausgewertet werden.

Ein LISP-Programm ist wie wir wissen eine Folge von Ausdrücken. Die Ausführung des Programms besteht darin, daß die einzelnen Ausdrücke nacheinander ausgewertet werden. Dies geschieht typischerweise in einem Interpretierer, der dazu

- jeweils einen Ausdruck einliest,
- den eingelesenen Ausdruck auswertet,
  dabei wird ein Wert ermittelt und möglicherweise ein Seiteneffekt bewirkt, und
- den ermittelten Wert ausgibt.

Dieser "Read-Eval-Print-Zyklus", auch Interpretationszyklus genannt, wird solange wiederholt, bis alle Ausdrücke des Programms ausgewertet sind.



Der erwähnte **Seiteneffekt** hat folgende Bedeutung:

Der Interpretierer ist zu jedem Zeitpunkt in einem Zustand, der die Gesamtheit der Zuordnungen von Atomen an Werte beschreibt.
Die Auswertung eines Ausdrucks bewirkt

- Ermittelung eines Wertes für diesen Ausdruck.
- Übergang vom bisherigen Zustand - etwa z - der vor Auswertung des Ausdrucks besteht, zu einem Zustand z , der sich von z durch neue und geänderte Zuordnungen unterscheiden kann.

Diese Änderung von Zuordnungen, oder kurz diese Änderung des Zustandes, haben wir als **Seiteneffekt** bezeichnet.
Bemerkenswert ist, daß in LISP die Änderung von Zuordnungen nur über Seiteneffekte möglich ist. Als einzige der bisher eingeführten Funktionen hat SETQ einen Seiteneffekt. Daß sie auch einen Wert liefert, wird gelegentlich benutzt, ist aber von untergeordneter Bedeutung.

Im Normalfall arbeitet man mit dem LISP-System interaktiv über die beschriebene Read-Eval-Print-Schleife. Man spricht auch von der **"Top-Level-Schleife"** oder kurz dem **Top-Level**; darin kommt zum Ausdruck, daß man sich auf der obersten Kontrollebene des Systems befindet.

2.4 Der Interpretierer als Systemfunktion

Der Interpretierer ist eine Systemfunktion, die automatisch auf jede eingegebene Form angewandt wird. Diese Systemfunktion kann auch explizit unter dem Funktionsnamen EVAL (von evaluate = auswerten) aufgerufen werden. Ein expliziter Aufruf von EVAL bewirkt eine "Extrarunde" in der Auswertung. Dies wird anschließend erläutert.

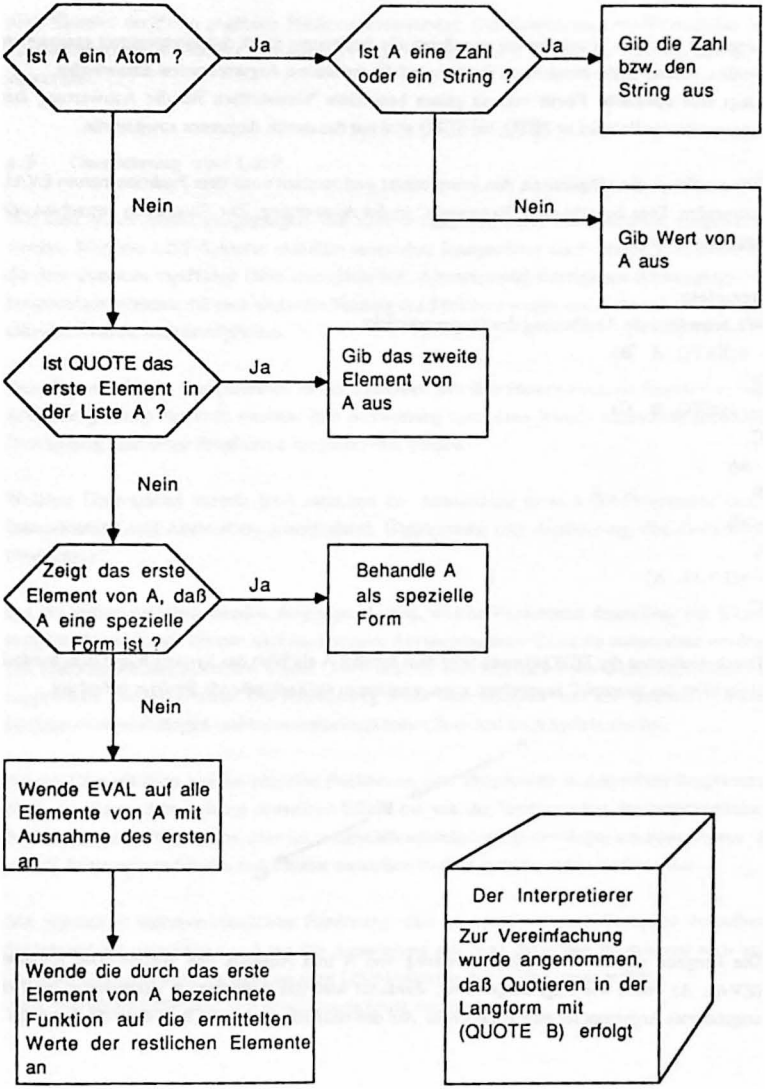
Eine detaillierte Beschreibung des Interpretierers und damit der Funktion EVAL kann hier noch nicht gegeben werden. Die Auswertung eines Ausdrucks A, der mit den uns jetzt bekannten Elementen der Sprache LISP gebildet ist, wird durch das nachfolgend angegebene Schema beschrieben.

Das Schema spiegelt folgende Möglichkeiten für den Aufbau eines Ausdrucks wieder:

Ein **Ausdruck** A ist entweder ein Atom oder eine Liste;

ein Atom ist	**entweder**	eine Zahl oder ein String diese haben sich selbst als Wert; bedürfen also keiner Auswertung
	oder	ein Symbol; dessen Wert muß durch Zuordnung im Speicher festgelegt sein
eine Liste ist	**entweder**	ein QUOTE-Ausdruck der Gestalt (QUOTE B); dieser hat den Wert B
	oder	ein Funktionsaufruf der Gestalt (<Funktionsbezeichnung> <Argument-1> ... <Argument-n>)

Die Funktionsbezeichnung läßt erkennen, ob eine "spezielle" Funktion (auch **"spezielle Form"** genannt) oder eine "normale" Funktion vorliegt.


Ist A ein Atom ?
Ja
Ist A eine Zahl oder ein String ?
Ja
Gib die Zahl bzw. den String aus
Nein
Nein
Gib Wert von A aus
Ist QUOTE das erste Element in der Liste A ?
Ja
Gib das zweite Element von A aus
Nein
Zeigt das erste Element von A, daß A eine spezielle Form ist ?
Ja
Behandle A als spezielle Form
Nein
Wende EVAL auf alle Elemente von A mit Ausnahme des ersten an
Wende die durch das erste Element von A bezeichnete Funktion auf die ermittelten Werte der restlichen Elemente an
Der Interpretierer
Zur Vereinfachung wurde angenommen, daß Quotieren in der Langform mit (QUOTE B) erfolgt

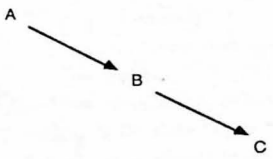
Liegt eine "normale" Funktion vor, so müssen alle Argumente durch den Interpretierer ausgewertet werden; sodann ist die bezeichnete Funktion auf die ermittelten Argumentwerte anzuwenden.
Liegt eine **spezielle Form** vor, so gelten besondere Vorschriften für die Auswertung der Argumente; ein Beispiel ist SETQ; bei SETQ wird nur das zweite Argument ausgewertet.

Wir erwähnten die Möglichkeit, den Interpretierer auch explizit unter dem Funktionsnamen EVAL aufzurufen. Dies bewirkt eine "Extrarunde" in der Auswertung. Zur Illustration betrachten wir folgende Beispiele:

Beispiele:
Wir betrachten die Abarbeitung des Programmstücks

```
-->(SETQ A 'B)
B
-->(SETQ B 'C)
C
->A
B
->B
C
-->(EVAL A)
C
```

Durch Auswerten der SETQ-Formen wird dem Symbol A als Wert das Symbol B und dem Symbol B als Wert das Symbol C zugeordnet; somit wird intern die nachstehende Struktur aufgebaut.



Die Eingabe von A veranlaßt Auswertung von A und Ausgabe des Wertes. Die Eingabe (EVAL A) wird wie folgt ausgewertet: Zunächst wird das Argument A ausgewertet, weil es unquotiertes Argument für eine Funktion ist. Auf den resultierenden Wert B ist dann die durch das

erste Element der Form gegebene Funktion anzuwenden. Die Anwendung von EVAL, also des Interpretierers auf B bedeutet Auswertung von B; dem Symbol B ist aber als Wert das Symbol C zugeordnet.

2.5 Übersetzung von LISP

Wir sind bisher davon ausgegangen, daß LISP-Programme durch Interpretation ausgewertet werden. Moderne LISP-Systeme enthalten neben dem Interpretierer noch weitere Komponenten, die dem Benutzer vielfältige Hilfe beim Erstellen, Austesten und Korrigieren (Debugging) von Programmen anbieten, für eine sinnvolle Nutzung des Speichers sorgen und nicht zuletzt auch eine effiziente Auswertung ermöglichen.

Eine der wichtigsten Komponenten ist der Compiler. Mit ihm können einzelne Funktionen oder ganze Programme übersetzt werden. Ihre Auswertung kann dann jeweils anhand der durch die Übersetzung erhaltenen Programme vorgenommen werden.

Welcher Unterschied besteht jetzt zwischen der Auswertung eines LISP-Programms durch Interpretation und einer Auswertung durch Übersetzung und Ausführung des übersetzten Programms ?

Bei der **Interpretation** werden diejenigen Listen, welche Funktionen darstellen, mit EVAL evaluiert. Diese Listen können aber i.a. erst nach Auswertung ihrer Elemente ausgewertet werden. Die Elemente können nun selbst wieder Listen sein, die auch erst nach Auswertung ihrer Elemente ausgewertet werden können. Die Auswertung durch Interpretation muß also (rekursiv) in den Listenstrukturen absteigen und hat entsprechend hohen Zeit- und auch Speicherbedarf.

Bei der **Compilation** werden einzelne Funktionen oder Programme in Assembler-Programme übersetzt, deren Abarbeitung denselben Effekt hat wie die Interpretation der ursprünglichen Funktionen bzw. Programme, aber i.a. wesentlich schneller ist (in der Regel um einen Faktor 10 bis 20). Interpretierer können andererseits wesentlich leichter erstellt werden als Compiler.

Die eigentlich selbstverständliche Forderung, daß Interpretierer und Compiler denselben Sprachumfang verarbeiten und bei der Auswertung ein- und desselben Programms stets auf dasselbe Resultat führen, ist nicht bei allen LISP-Dialekten und LISP-System erfüllt.
Bei **COMMON LISP** sind hier keine Unterschiede zugelassen.

2.6 Zusammenfassung

LISP-Daten und LISP-Programme sind aufgebaut aus Ausdrücken.

Ausdrücke sind Atome oder Listen, die Elemente von Listen sind selbst wieder Ausdrücke.

Atome sind Zahlen, Strings (diese haben sich stets selbst als Wert) oder **Symbole** (diese haben einen Wert, wenn ihnen ein solcher zugeordnet ist). Mit SETQ kann einem Symbol ein Wert zugewiesen werden.
Listen haben einen Wert, wenn ihre Bestandteile einen Wert haben.

Ein **Programm** ist eine Folge von Ausdrücken; es wird ausgewertet, indem die Ausdrücke der Reihe nach vom Interpretierer ausgewertet werden.

Die **Auswertung** eines Ausdrucks kann den Zustand des Interpretierers verändern; man spricht dann von einem Seiteneffekt. Die **Wertzuweisung** mit SETQ beruht auf einem derartigen Seiteneffekt.

Der Interpretierer kann selbst mit EVAL in einem Ausdruck aufgerufen werden; dies bewirkt eine "Extrarunde" in der Auswertung.

Auch Sonderwünsche sind erfüllbar:
Ein zusätzlicher Auswertungsschritt wird vorgenommen, wenn man den Interpretierer explizit aufruft; EVAL macht's möglich.

Die Auswertung eines Ausdrucks unterbleibt, wenn man ihn in QUOTE verpackt oder - wie man sagt - quotiert. Pro Auswertungsversuch wird eine Quotierung verbraucht:
Der Wert eines quotierten Ausdrucks ist der Ausdruck selbst.

Häufig benutzte Funktionen werden meist durch den Compiler übersetzt, da sie dann wesentlich schneller abgearbeitet werden können.

KAPITEL 3:

Aufbau und Zerlegung von Listen

Die besondere Stärke von LISP liegt nicht in der Durchführung arithmetischer Berechnungen, sondern in der Verarbeitung von symbolischen Ausdrücken. Als Grundlage hierfür behandelt dieses Kapitel jetzt elementare Operationen für Aufbau und Zerlegung von Listen.

3.1 Aufbau von Listen mit CONS

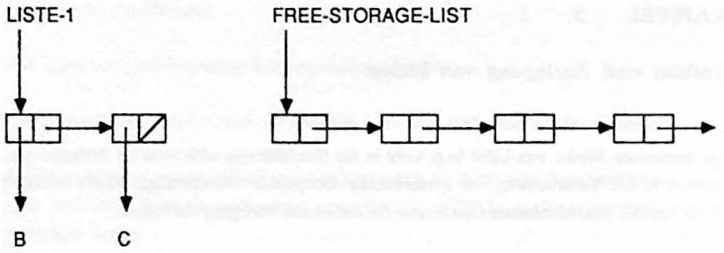
Zum Aufbau von Listen verwendet man insbesondere die Symstemfunktion CONS. Sie wird aufgerufen mit dem
Aufrufmuster:

(CONS <Ausdruck> <Ausdruck>)

Dabei behandeln wir zunächst nur den Fall, daß das zweite Argument als Wert eine Liste hat.
In diesem Fall liefert CONS als
Wert:
Diejenige Liste, die entsteht, wenn man den Wert des ersten Ausdrucks als zusätzliches Element vorne an die durch das zweite Argument gegebene Liste anfügt.
Seiteneffekt: Keiner

Beispiel:
-->(CONS 'A '(B C))
(A B C)

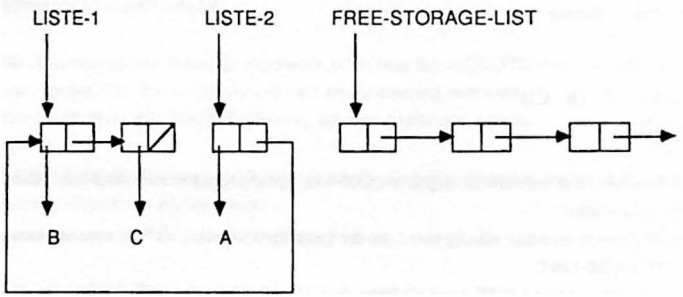
Um zu verstehen, was bei Anwendungen von CONS im Speicher des Rechners geschieht, müssen wir folgendes wissen:
Jedes LISP-System unterhält ständig eine Liste der freien Speicherzellen, als Wert etwa des Atoms FREE-STORAGE-LIST.
Angenommen das Atom LISTE-1 hat als Wert die Liste (B C); dann enthält der Speicher eine Struktur gemäß nachstehendem Diagramm.



Jetzt betrachten wir die Auswertung von

(SETQ LISTE-2 (CONS 'A LISTE-1))

Dazu wird der Wert von (CONS 'A LISTE-1) ermittelt und dem Symbol LISTE-2 als Wert zugewiesen. Im Speicher geschieht dabei folgendes:
Vom Kopfende der Liste FREE-STORAGE-LIST wird eine freie Speicherzelle entfernt und mit neuen Zeigern ausgerüstet; der linke Zeiger wird auf das Atom A gerichtet; der rechte auf die erste Speicherzelle der Liste-1. Damit ist die Verbindungsstruktur der Liste (A B C) aufgebaut. Gleichzeitig wird ein Zeiger vom Atom LISTE-2 auf den Kopf der Liste (A B C) gerichtet. Damit liegt folgende Struktur vor



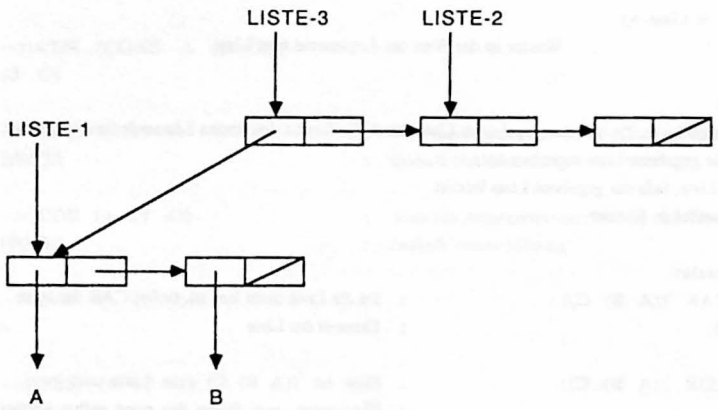
Weitere Beispiele:

```
-->(CONS '(A B) '(C D))
((A B) C D)
```

Beim Abarbeiten des Programmstücks

```
-->(SETQ LISTE-1 '(A B))
(A B)
-->(SETQ LISTE-2 '(C D))
(C D)
-->(SETQ LISTE-3 (CONS LISTE-1 LISTE-2))
((A B) C D)
```

werden die angegebenen Ausgaben erzeugt und folgende interne Struktur aufgebaut:



3.2 Zerlegung von Listen mit CAR und CDR

Die Bausteine, aus denen eine Liste beim Aufbau mit CONS aufgebaut wird, lassen sich mit Hilfe der Systemfunktionen CAR und CDR zurückgewinnen:
CAR liefert bei Anwendung auf eine beliebige Liste das erste Listenelement.

CDR liefert bei Anwendung auf eine beliebige Liste die nach Wegnahme des ersten Listenelements verbleibende Liste.

Aufrufmuster für CAR:

(CAR <-Liste->)

Die Niederschrift des Arguments in der Gestalt <-Liste-> bedeutet, daß der Wert dieses Arguments eine Liste sein soll.

Wert:
Erstes Element der gegebenen Liste, falls diese nicht leer ist.
Leere Liste ( ), falls die gegebene Liste leer ist.
Seiteneffekt: Keiner

Aufrufmuster für CDR:
(CDR <-Liste->)

Wieder ist der Wert des Arguments eine Liste.

Wert:
Diejenige Liste, die aus der gegebenen Liste durch Entfernen des ersten Listenelements entsteht, falls die gegebene Liste nicht leer ist.
Leere Liste, falls die gegebene Liste leer ist.
Seiteneffekt: Keiner

Beispiele:

```
-->(CAR '((A B) C))                ; Da die Liste nicht leer ist, liefert CAR das erste
(A B)                              ; Element der Liste

-->(CDR '((A B) C))                ; Hier ist ((A B) C) eine Liste mit zwei
(C)                                ; Elementen, von denen das erste selbst wieder
                                   ; eine Liste ist.


-->(SETQ SPRUCH '(WISSEN IST MACHT))
(WISSEN IST MACHT)

-->(CAR SPRUCH)
WISSEN
```

```
-->(CDR SPRUCH)
(IST MACHT)

-->(CAR 'SPRUCH)                    ; Wert des Arguments von CAR ist ein Atom,
ERROR                               ; deshalb Fehlermeldung

-->(CDR 'SPRUCH)                    ; Wert des Arguments von CDR ist ein Atom,
ERROR                               ; deshalb Fehlermeldung

-->(CDR '(CAR '((A B) C)))          ; Quotierung schützt (CAR '((A B) C))
((QUOTE ((A B) C)))                 ; vor Auswertung

-->(CAR (CONS 'A '(B C)))
A

-->(CDR (CONS 'A '(B C)))
(B C)

-->(CAR (+ 17 4))                   ; Wert des Arguments von CAR ist das Atom 21,
ERROR                               ; deshalb Fehlermeldung

-->(CDR (+ 17 4))                   ; Wert des Arguments von CDR ist das Atom 21,
ERROR                               ; deshalb Fehlermeldung

-->(CAR '(+ 17 4))
+

-->(CDR '(+ 17 4))
(17 4)
```

Durch eine geeignete Folge von Anwendungen der Funktionen CAR und CDR können Teilausdrücke aus dem Innern einer geschachtelten Liste herausgegriffen werden. Die umständliche Niederschrift einer Mehrfachanwendung wie

```
(CAR (CDR (CAR (CAR LISTE))))
```

kann abgekürzt werden zu

```
(CADAAR LISTE)
```

Beispiel:

```
-->(CDDAAR '(((A B (C D)) E) F (G H)))
((C D))
```

Die Anzahl der Anwendungen von CAR und CDR, die auf diese Weise zusammengefaßt werden können, ist auf 4 beschränkt.

Bei Anwendung auf Listen wird der Zusammenhang zwischen CONS einerseits und den Funktionen CAR und CDR andererseits durch folgendes Schema verdeutlicht:

(CONS $'\Pi$ $'(\Sigma_1 \ldots \Sigma_n)$) hat Wert $(\Pi \ \Sigma_1 \ldots \Sigma_n)$

(CAR $'(\Pi \ \Sigma_1 \ldots \Sigma_n)$) hat Wert Π

(CDR $'(\Pi \ \Sigma_1 \ldots \Sigma_n)$) hat Wert $(\Sigma_1 \ldots \Sigma_n)$

3.3 Mehr zu CONS, CAR und CDR

Die Funktion CONS kann allgemeiner eingesetzt werden als bisher gezeigt wurde. Wir haben zu ihrem **Aufrufmuster**

(CONS <Ausdruck> <Ausdruck>)

bisher nur den Fall betrachtet, daß der Wert des zweiten Arguments eine Liste ist. In diesem Fall liefert auch die Auswertung stets eine Liste.

Das **Beispiel**

```
-->(CONS 'A 'NIL)
(A)
```

ist bekannt. Nun verwenden wir als zweites Argument an Stelle von NIL ein Atom:

```
-->(CONS 'A 'B)
(A . B)
```

In der Ausgabe erscheint jetzt ein Paar mit den Werten der beiden Argumente, aber getrennt durch einen Punkt und je ein Leerzeichen links und rechts desselben. Wir sprechen dabei von einem **CONS-Paar**, auch von einem Punkt-Paar (englisch "dotted pair").

Grundsätzlich liefert nämlich **jede Auswertung eines Aufrufs** der Gestalt

(CONS <Ausdruck-1> <Ausdruck-2>)

als **Wert ein CONS-Paar** der "Bauart"

(<Wert des ersten Ausdrucks> . <Wert des zweiten Ausdrucks>)

Ist die 2. Komponente eines solchen CONS-Paares selbst wieder ein CONS-Paar, so werden i.a. zur Abkürzung der Punkt des äußeren und die Klammern des CONS-Paares weggelassen:

```
-->(CONS 'A (CONS 'B 'C))
(A B . C)                              ; anstatt (A . (B . C))
```

Bei Eingabe von Daten kann diese Abkürzung nach Belieben verwendet werden. In der Ausgabe durch das System wird sie immer benutzt.

Wir betrachten noch speziell CONS-Paare der "Bauart"

(A-1 . (A-2 . (A-3 . (... . (A-n . B) ...))))

mit einem Atom B. Ihre abgekürzte Darstellung besteht

im Falle B ungleich NIL aus der "Punkt-Liste" (dotted list)
(A-1 A-2 A-3 ... A-n . B)

im Falle B = NIL aus der "Liste"
(A-1 A-2 A-3 ... A-n).

Die bisher eingeführten Listen erweisen sich jetzt als spezielle CONS-Paare.

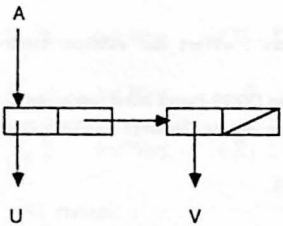
Die Systemfunktionen CAR und CDR liefern bei Anwendung auf ein beliebiges CONS-Paar die erste bzw. zweite Komponente. Bei Einschränkung auf solche CONS-Paare, die Listen darstellen, entspricht dies der eingeführten Bedeutung von CAR und CDR für Listen.

Die Speicherstrukturen für "Punkt-Listen" und "Listen" unterscheiden sich wie folgt:

Bei Auswertung von

```
-->(SETQ A '(U V))
(U V)
```

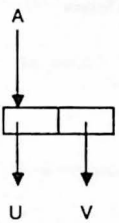
wird intern die uns bekannte Struktur aufgebaut:



Bei Auswertung von

```
-->(SETQ A '(U . V))
(U . V)
```

wird intern eine Struktur wie folgt aufgebaut:



Dies zeigt:

> Sieht man Listen und nicht CONS-Paare als grundlegende Datenstruktur in LISP, so gewinnt man an Übersicht und Einheitlichkeit in der externen Darstellung auf Kosten von etwas höherem Speicherbedarf.

Wir halten fest:
Unter Verwendung von CONS-Paaren kann verschiedentlich etwas speichereffizienter programmiert werden. Dies kann sinnvoll sein, wenn Paare von Atomen verarbeitet werden müssen.
Es gibt erfahrene LISP-Programmierer, die nur Listen und keine allgemeinen CONS-Paare verwenden und damit erfolgreich und glücklich sind.

3.4 Zusammenbau von Listen mit APPEND und LIST

Die bisher eingeführten Funktionen **CONS, CAR** und **CDR** reichen zwar im Prinzip zum Aufbau beliebiger Listen aus. Um weniger umständlich (und damit auch zuverlässiger) programmieren zu können, gibt es noch verschiedene Funktionen zum Aufbau von Listen; wir führen davon zunächst APPEND und LIST ein.

Mit Hilfe der Funktion APPEND können mehrere Listen zu einer einzigen zusammengefaßt werden. APPEND kann mit zwei oder mehr Argumenten aufgerufen werden. Wir betrachten hier nur den Fall, daß diese Argumente als Werte Listen haben. Dann fügt APPEND die Elemente dieser Listen in der vorgegebenen Reihenfolge zu einer neuen Liste zusammen.
Die Funktion LIST erwartet als Argumente beliebige Ausdrücke und bildet aus den Werten der Argumente eine Liste.

Aufrufmuster für APPEND:

(APPEND <-Liste-1-> <-Liste-2-> . . . <-Liste-n->)

Hier bedeutet <-Liste-i-> für $1 \leq i \leq n$ jeweils ein Argument, dessen Wert eine Liste ist.

Wert:
Diejenige Liste, die durch Aneinanderfügen der gegebenen Listen entsteht.
Seiteneffekt: Keiner

Aufrufmuster für LIST:

(LIST <Ausdruck-1> <Ausdruck-2> ... <Ausdruck-n>)

Wert:
Diejenige Liste, welche die Werte der Argumente in der gegebenen Reihenfolge als Elemente enthält.
Seiteneffekt: Keiner

Die Unterschiede zwischen APPEND, LIST und CONS werden illustriert durch

Beispiele:

```
-->(CONS '(A B) '(C D))
((A B) C D)

-->(APPEND '(A B) '(C D)
(A B C D)

-->(LIST '(A B) '(C D))
((A B) (C D))

-->(SETQ L '(A B))
(A B)

-->(CONS L L)
((A B) A B)

-->(APPEND L L)
(A B A B)

-->(LIST L L)
((A B) (A B))

-->(CONS 'L L)
(L A B)

-->(APPEND 'L L)
ERROR

-->(LIST 'L L)
(L (A B))
```

```
 -->(APPEND '(C) '( ) L '(D E))
(C A B D E)

-->(LIST L L '(C D))
((A B) (A B) (C D))

-->(APPEND '( ) '( ))
NIL

-->(LIST '( ) '( ))
(NIL NIL)
```

3.5 LIST*, eine Modifikation von LIST

Wir haben gesehen, wie die Funktion LIST die Werte ihrer Argumente zu einer Liste zusammenfaßt.
Zuerst wird mit dem letzten Argument und NIL ein CONS-Paar gebildet, an dieses werden die weiteren Argumente "angeCONSt".

Beispiel:

```
-->(LIST 'A1 ... 'X 'Y 'Z)
(A1 ... X Y Z)                    ; als Abkürzung für
                                  ; (A1 . ... (X . (Y . (Z . NIL)))...)
```

Wir betrachten jetzt eine Modifikation von LIST, nämlich die Funktion LIST*. Sie hat dasselbe **Aufrufmuster** wie LIST, nämlich

(LIST* <Ausdruck-1> ... <Ausdruck-n>)

Wert:
Diejenige Punktliste, die aus dem CONS-Paar, das mit dem vorletzten und letzten Argument gebildet wird, durch "AnCONSen" der weiteren Argumente hervorgeht.

Beispiel:

```
-->(LIST* 'A1 ... 'X 'Y 'Z)
(A1 ... X Y . Z)                  ; als Abkürzung für
                                  ; (A1 . ... (X . (Y . Z))...)
```

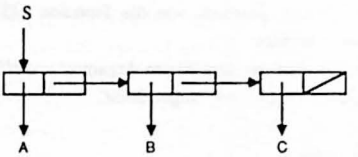
Ist das letzte Argument von LIST* kein Atom, sondern eine Liste, so bewirkt LIST* nur, daß die Werte der restlichen Argumente an diese Liste vorne angefügt werden.

Beispiel:

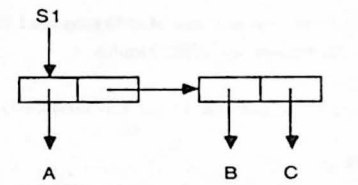
```
-->(LIST* 'A 'B 'C '(D E F))
(A B C D E F)

-->(LIST* 'A '(B C) '(D E F))
(A (B C) D E F)
```

Zwei weitere **Beispiele**
zeigen die Unterschiede der internen Strukturen bei LIST und LIST*:

```
-->(SETQ S (LIST 'A 'B 'C))      ; eingerichtet wird die Struktur
(A B C)                          ;
```




```
-->(SETQ S1 (LIST* 'A 'B 'C))    ; jetzt wird folgende Struktur eingerichtet
(A B . C)
```



3.6 Garbage-Collection

Wir haben gesehen, wie die Funktion CONS die zum Aufbau von Listen benötigten Speicherzellen von der FREE-STORAGE-LIST holt. Ähnlich läuft die Beschaffung neuer Speicherzellen bei Anwendung von APPEND, LIST und LIST*.

Die Anzahl der freien Speicherzellen ist jedoch nicht unbeschränkt. Andererseits werden viele Speicherstrukturen schon bald nach dem Aufbau nicht mehr benötigt. Wir betrachten ein einfaches

Beispiel:

```
-->(SETQ WASTE '(A B C D))
(A B C D)
    .
    .
-->(SETQ WASTE (LIST 'X 'Y))
(X Y)
```

Nach Abarbeitung dieses Programmstücks ist die Struktur zur Liste (A B C D) zwar noch vorhanden, aber nicht mehr zugänglich. Deshalb werden während der Abarbeitung von LISP-Programmen von Zeit zu Zeit die nicht mehr zugänglichen Speicherzellen ermittelt und wieder in die FREE-STORAGE-LIST eingereiht.
Diese Aufgabe wird ohne Zutun des Benutzers vom sogenannten **Garbage-Collector** erledigt. Der entsprechende Vorgang heißt "Carbage-Collection". Eine allgemein anerkannt deutsche Übersetzung gibt es dafür nicht.

3.7 Zusammenfassung

Listen können mit der Systemfunktion CONS aufgebaut, mit den Systemfunktionen CAR und CDR abgebaut werden.

Mit CONS wird an eine gegebene Liste vorne ein Element als neuer Listenkopf angefügt oder "angeCONSt"; Anwendung von CAR bzw. CDR auf eine Liste liefert den Listenkopf bzw. den Listenrest ohne Listenkopf.
CONS kann nicht nur zum Aufbau von Listen eingesetzt werden. Im Grunde baut es CONS-Paare, wir sprechen auch von Punkt-Paare (dotted pairs) oder Punkt-Listen (dotted lists) auf, welche die bisher eingeführten "normalen" Listen als Spezialfälle beinhalten.
CAR und CDR liefern umgekehrt wieder die Komponenten dieser CONS-Paare.

APPEND fügt beliebig viele Listen zusammen.
LIST listet seine Argumente in einer neu geschaffenen Liste auf, indem es aus dem letzten Argument und NIL ein CONS-Paar bildet und die weiteren Argumente vorne "anCONSt".

LIST* leistet fast dasselbe wie LIST, beginnt aber mit dem Aufbau eines CONS-Paares aus dem vorletzten und letzten Argument und "CONSt" daran die weiteren Argumente an.
Durch Garbage-Collection werden die nicht mehr zugänglichen Strukturen ermittelt und die involvierten Speicherzellen zur erneuten Verwendung gesammelt.

KAPITEL 4:

Prädikate

Die Auswertung von Funktionen oder Teilen eines Programms muß oftmals von Bedingungen abhängig gemacht werden. Bedingungen werden mit Hilfe von Prädikaten überprüft. Ein Prädikat ist eine Funktion, die als Wert nur einen der Wahrheitswerte "wahr" und "falsch" annehmen kann. Der Wert "wahr" bzw. "falsch" zeigt an, daß die Argumente der Funktion eine Bedingung erfüllen bzw. nicht erfüllen. In LISP wird "falsch" stets durch das Symbol NIL und "wahr" durch das Symbol T dargestellt; aber auch jeder Wert ungleich NIL wird in Positionen, in denen ein Wahrheitswert erwartet wird, als Ersatz für "wahr" angesehen. Die Symbole T und NIL sind spezielle Symbole; sie haben (wie Zahlen) sich selbst als Wert.

Wir lernen in diesem Kapitel einige fundamentale Testprädikate und Testfunktionen kennen. Sie betreffen die Prüfung,

- ob ein Ausdruck ein Atom, ein CONS-Paar bzw. eine Liste ist,
- ob vorgegebene LISP-Strukturen übereinstimmen,
- ob ein Ausdruck in einer Liste als Element auftritt oder nicht.

4.1 Drei grundlegende Prädikate: ATOM, CONSP und LISTP

Die grundlegenden Typen von LISP-Objekten sind sicherlich Atome, CONS-Paare und Listen. Prädikate für entsprechende Typprüfungen sind deshalb besonders wichtig. Sie werden im folgenden behandelt.

Wir konnten den Begriff des Atoms zunächst nur anhand der wichtigsten Beispiele, nämlich Zahlen, Symbole und Strings intuitiv einführen, sind aber jetzt in der Lage, eine exakte Definition zu geben:

In COMMON LISP wird jedes Datenobjekt, das nicht mit CONS aufgebaut werden kann, als **Atom** bezeichnet.

Dies unterstreicht die fundamentale Bedeutung der Aufbaufunktion CONS; jedes Datenobjekt ist entweder ein Atom oder ein CONS-Paar. Die Prädikate für die zugehörigen Typprüfungen sind

ATOM und CONSP

ATOM testet, ob der Wert seines Arguments ein Atom ist.
CONSP testet, ob der Wert seines Arguments mit CONS aufgebaut werden kann.

Aufrufmuster für ATOM:

(ATOM <Ausdruck>)

Wert:
T falls der Wert des Arguments nicht mit CONS aufgebaut werden kann
NIL sonst
Seiteneffekt: Keiner

Aufrufmuster für CONSP:

(CONSP <Ausdruck>)

Wert:
T falls der Wert des Ausdrucks mit CONS aufgebaut werden kann
NIL sonst
Seiteneffekt: Keiner

Beispiele:

```
-->(ATOM 'NIL)
T

-->(ATOM 'T)
T

-->(ATOM 'ZWILLINGE)
T

-->(SETQ ZWILLINGE '(MAX MORITZ))
(MAX MORITZ)

-->(ATOM ZWILLINGE)
NIL
```

```
-->(CONSP ZWILLINGE)
T

-->(ATOM '(MAX MORITZ))
NIL

-->(CONSP '(MAX MORITZ))
T

-->(ATOM (+ 17 4))                ; Wert des Arguments ist die Zahl 21 , sie ist
T                                 ; ein Atom

-->(ATOM '(+ 17 4))               ; Wert des Arguments ist eine nichtleere
NIL                               ; Liste, sie wird aufgebaut durch
                                  ; (CONS '+ '(17 4))

-->CONSP '(+ 17 4))
T

-->(CONSP (CONS 'A 'B))           ; Wert des Arguments ist das CONS-Paar
T                                 ; (A . B)

-->(CONSP '(A B C))               ; Wert des Arguments ist die Liste (A B C),
T                                 ; aufgebaut durch (CONS 'A '(B C))

-->(CONSP '(MAX . MORITZ))
T

-->(CONSP 'MAX)
NIL

-->(CONSP '17)
NIL

-->(CONSP 'NIL)                   ; NIL ist nicht als Wert einer CONS-Form
NIL                               ; erzeugbar
```

Wir kommen nun zum Prädikat LISTP; es testet, ob eine "allgemeine Liste" vorliegt im Sinne der leeren Liste oder einer mit CONS aufgebauten Liste bzw. Punkt-Liste.

Aufrufmuster für LISTP:

(LISTP <Ausdruck>)

Wert:

T	falls der Wert des Arguments das Prädikat CONSP erfüllt oder gleich NIL ist.
NIL	sonst.

Seiteneffekt: Keiner

LISTP und CONSP unterscheiden also nicht zwischen echten Listen (abgeschlossen durch NIL) und Punkt-Listen (abgeschlossen durch ein Atom ungleich NIL).

Beispiele:

```
-->(SETQ ZWILLINGE '(MAX MORITZ))
(MAX MORITZ)

-->(LISTP ZWILLINGE)
T

-->(LISTP 'ZWILLINGE)
NIL

-->(LISTP '(MAX MORITZ))
T

-->(LISTP '(A . B))
T

-->(LISTP (CONS 'A 'B))
T

-->(LISTP 'T)
NIL
```

```
-->(LISTP '( ))                    ; leere Liste erfüllt LISTP
T

-->(LISTP 'NIL))
T
```

NIL ist der einzige Ausdruck, der sowohl das Prädikat ATOM wie das Prädikat LISTP erfüllt.

4.2 Übereinstimmung von LISP-Strukturen

Im folgenden betrachten wir Prädikate, mit denen die Übereinstimmung von LISP-Strukturen getestet werden kann. Offensichtlich gibt es mindestens zwei Möglichkeiten eine Übereinstimmung festzulegen:

1) Identität der internen Speicherstrukturen, kurz der internen Darstellung.
2) Übereinstimmung der von einem Ausgabegerät ausgegebenen Darstellungen, kurz der externen Darstellung.

Die entsprechenden Tests können vorgenommen werden mit den Prädikaten EQ und EQUAL.

Aufrufmuster von EQ:
(EQ <Ausdruck> <Ausdruck>)

Wert:
T falls den beiden Argumenten ein- und dieselbe Speicherstruktur zugrundeliegt.
NIL sonst.

Seiteneffekt: Keiner

Ein Symbol ist immer "EQ" zu sich selbst, da es intern stets durch ein- und denselben Speicherbereich dargestellt wird. Damit gilt:

```
-->(EQ 'A 'A)
T
-->(EQ 'A 'B)
NIL
```

Aufrufmuster von EQUAL:
(EQUAL <Ausdruck> <Ausdruck>)

Wert:

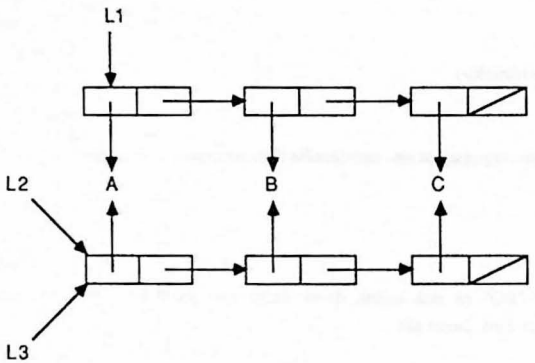
T	falls die beiden Ausdrücke nach Ausgabe über ein Ausgabemedium gleich aussehen
NIL	sonst.

Seiteneffekt: Keiner

Beispiele:
Bei Auswertung der nachfolgenden SETQ-Formen

```
-->(SETQ L1 (LIST 'A 'B 'C))
(A B C)
-->(SETQ L2 (LIST 'A 'B 'C))
(A B C)
-->(SETQ L3 L2)
(A B C)
```

wird folgende interne Struktur aufgebaut:



Bei jeder Auswertung von LIST wird nämlich mit neuen Zellen von der FREE-STORAGE-LIST eine neue Liste aufgebaut. So haben L1 und L2 zwar denselben Wert (A B C), werden aber

durch verschiedene Strukturen dargestellt. Diese Strukturen stimmen nur in den Symbolen A, B und C überein. Dementsprechend werden die nachfolgenden Funktionsaufrufe wie angegeben ausgewertet.

```
-->(EQ L1 L2)
NIL
-->(EQ L2 L3)
T
-->(EQUAL L1 L2)
T
-->(EQUAL L2 L3)
T
-->(EQ (CAR L1) (CAR L2))          ; (CAR L1) und (CAR L2) haben
T                                  ; beide als Wert das Atom A
```

Der letzte Aufruf von EQ macht deutlich:
Mit Hilfe von EQ kann auch festgestellt werden, ob zwei insgesamt verschiedene Strukturen gemeinsame Teile haben.

Zur Verwendung der Prädikate EQ und EQUAL sei noch auf folgendes hingewiesen:

- Bei Übereinstimmung von Ausdrücken bezüglich EQ besteht auch Übereinstimmung bezüglich EQUAL. Die Umkehrung gilt nicht.
- Die Bedeutung von EQ liegt vor allem darin, daß es effektiver getestet werden kann als EQUAL: Es ist einfacher für zwei Strukturen festzustellen, ob sie identisch sind oder nicht (Test mit EQ), als festzustellen, ob eine Struktur eine Kopie der anderen ist (Test mit EQUAL).

4.3 Vergleich von Zahlen

Für Zahlen ist das Ergebnis der Anwendung von EQ in COMMON LISP nicht definiert, da Zahlen auch desselben Typs (man unterscheidet ganze Zahlen, Gleitpunktzahlen verschiedener Genauigkeit, komplexe Zahlen u.a.) und desselben Wertes nicht notwendigerweise für jedes Auftreten im selben Speicherbereich untergebracht werden. Zahlen desselben Typs können mit EQUAL verglichen werden. EQUAL vergleicht die externe Darstellung seiner Argumente und diese ist bei Zahlen vom gleichen Typ und Wert identisch.

Beispiele:

```
-->(EQ 2.0 2)                          ; Ergebnis in COMMON LISP nicht definiert
NIL                                    ; Da Interndarstellung sicher verschieden,
                                       ; im allgemeinen Wert NIL

-->(EQ 2.13 2.13)                      ; Ergebnis in COMMON LISP nicht
NIL oder T                             ; definiert, Antwort ist systemabhängig

-->(EQUAL 2.0 2)                       ; Zahlen unterschiedlichen Typs
NIL

-->(EQUAL 4.5 (+ 2.3 2.2))             ; Übereinstimmung im Typ
T

-->(EQUAL 4.5 '(+ 2.3 2.2))            ; Quotierung verhindert Auswertung
NIL
```

Für den Vergleich von Zahlen ist also EQ **absolut ungeeignet** und EQUAL **nur bedingt geeignet.**

Zum Vergleich von Zahlen stellt COMMON LISP spezielle Prädikate bereit:
Wir betrachten zunächst = , < und > (gesprochen "gleich", "kleiner", "größer"):
Sie testen für beliebig viele Zahlen

- ob diese alle denselben Wert haben (Test mit =),
- ob diese aufsteigend geordnet sind (Test mit <),
- ob diese absteigend geordnet sind (Test mit >).

Aufrufmuster:

(= <-Zahl-1-> ... <-Zahl-n->)

(< <-Zahl-1-> ... <-Zahl-n->)

(> <-Zahl-1-> ... <-Zahl-n->) jeweils mit $n \geq 1$

Das bedeutet, daß jedes dieser Prädikate beliebig viele Argumente haben kann, wobei die Werte aller Argumente Zahlen sein müssen.

Bei Aufruf mit Argumenten, deren Werte **keine Zahlen** sind, erfolgt grundsätzlich eine **Fehlermeldung.**

Bei Aufruf mit $n \geq 2$ Argumenten, deren Werte Zahlen sind, erhält man den
Wert:

T falls alle Zahlen
- denselben Wert haben für =
- verschiedene Werte haben und aufsteigend geordnet sind für <
- verschiedene Werte haben und absteigend geordnet sind für >

NIL sonst.

Bei Aufruf mit nur einem Argument wird grundsätzlich der Wert T übergeben.

Seiteneffekt:
=, < und > haben selbst keine Seiteneffekte, aber die Auswertung der Argumente kann Seiteneffekte haben.

Beispiele:

```
-->(= 2.0 2.00 2)
T

-->(< -1.8 0 2.3 3)
T                                   ; Zahlenwerte streng aufsteigend geordnet

-->(> 17)                           ; Zahl 17 als einziges Argument
T

-->(= 17.0 (SETQ X (+ 13 4)))       ; Seiteneffekt wegen SETQ
T
```

COMMON LISP kennt zum Vergleich von Zahlen neben den Prädikaten =, <, > noch die Prädikate /=, <= und >= (gesprochen "ungleich", "kleiner-gleich" und "größer-gleich") mit Aufrufmustern wie oben, also mit $n \geq 1$ Argumenten, deren Werte Zahlen sind.

Im Falle $n = 1$ wird wieder grundsätzlich der Wert T übergeben.
Im Falle $n \geq 2$ ergibt sich als

Wert:

T falls alle Zahlen
- verschiedene Werte haben für /=
- monoton aufsteigend geordnet sind für <=
- monoton absteigend sind für >=

NIL sonst

Beispiele:

```
-->(/= 4 7 3 2)                    ; alle Zahlen verschieden
T

-->(/= 4 7 3 7)                    ; nicht alle verschieden
NIL

-->(>= 7 6 6 3 2)
T
```

Seiteneffekt:

Auch /= , >= und >= haben selbst keine Seiteneffekte; allerdings kann die Auswertung der Argumente Seiteneffekte haben.

4.4 Mehr zum Testen auf Übereinstimmung

Neben den Prädikaten EQ und EQUAL kennt COMMON LISP noch das Prädikat EQL; es vergleicht Strukturen genauso wie EQ und Zahlen genauso wie EQUAL.

Aufrufmuster:

(EQL <Ausdruck> <Ausdruck>)

Wert:

T falls beide Ausdrücke
entweder Zahlen desselben Typs und desselben Wertes sind
oder durch ein- und dieselbe Speicherstruktur dargestellt werden.

NIL sonst.

Seiteneffekt: Keiner

An dieser Stelle muß auf eine wichtige Äquivalenz hingewiesen werden:
NIL und die leere Liste - wir stellen sie durch ( ) dar - sind äquivalent.
Der Wert von (EQ NIL '( )) ist T.

Die leere Liste wird bei der Ausgabe stets durch NIL dargestellt. Im Zusammenhang damit ist auch festzuhalten, daß (ATOM '( )) stets den Wert T hat, da NIL als Atom gilt. Dementpsrechend hat der Ausdruck (ATOM S), unter der Annahme, daß S an eine Liste gebunden ist, genau dann den Wert T, wenn diese Liste leer ist. Nichtbeachtung führt zu rätselhaften Fehlern. Übrigens ist die leere Liste der einzige Ausdruck, der zugleich Liste und Atom ist.

4.5 Mitgliedschaft in einer Liste, die Funktion MEMBER

Ob ein Ausdruck in einer Liste als Listenelement auftritt, kann mit Hilfe der Funktion MEMBER überprüft werden.

Aufrufmuster:
(MEMBER <Ausdruck> <-Liste->)

Das bedeutet: Das erste Argument ist ein Ausdruck, das zweite Argument hat als Wert eine Liste.

Wert:
Die Liste wird daraufhin durchsucht, ob ein Listenelement (und zwar auf dem obersten Niveau !) mit dem gegebenen Ausdruck im Wert übereinstimmt. Wird kein solches Listenelement gefunden, so wird NIL übergeben. Andernfalls wird der Rest der Liste, beginnend mit dem ersten übereinstimmenden Listenelement, übergeben.
Seiteneffekt: Keiner

Beispiel:

```
-->(SETQ FUENF-X '(X1 X2 X3 X4 X5))
(X1 X2 X3 X4 X5)

-->(MEMBER 'X6 FUENF-X)
NIL

-->(MEMBER 'X3 FUENF-X)
(X3 X4 X5)
```

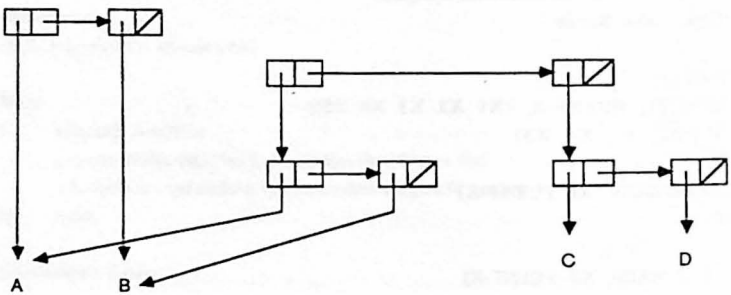
```
-->(SETQ A 'X4)
X4

-->(MEMBER A FUENF-X)
(X4 X5)

-->(MEMBER A '((X1 X2) (X3 X4 X5)))
NIL                                ; X4 ist nicht Element der zweielementigen
                                   ; Liste

-->(MEMBER A (CDR '((X1 X2) (X3 X4 X5))))
NIL                                ; X4 ist nicht Element der Liste
                                   ; ((X3 X4 X5))

-->(MEMBER A (CADR '((X1 X2) (X3 X4 X5))))
(X4 X5)                            ; X4 ist Element der Liste   (X3 X4 X5)
```

Bei der Auswertung der Funktion MEMBER liegt üblicherweise der Gleichheitstest mit EQL zugrunde. Dies wirkt sich nicht aus, solange der Wert des ersten Arguments von MEMBER ein Symbol ist, kann aber zu Überraschungen führen wie im folgenden

Beispiel:

```
-->(MEMBER '(A B) '((A B) (C D)))
NIL
```

Hier wird folgende interne Struktur zugrundegelegt:



Sodann wird überprüft, ob die Struktur zur Liste (A B) ein Teil der Struktur zur Liste ((A B) (C D)) ist. Dies ist nicht der Fall; somit wird NIL ausgegeben, obwohl (A B) ein Element der Liste ((A B) (C D)) ist. Ursache ist die Verwendung von EQL für den Gleichheitstest in MEMBER. Da die interne Struktur für (A B) nicht Teil der internen Struktur von ((A B) (C D)) ist, wird (A B) nicht als Listenelement erkannt.

Es ist möglich, für den Gleichheitstest in MEMBER die Verwendung von EQUAL vorzuschreiben. Dazu gibt man beim Aufruf von MEMBER in der Position eines dritten Arguments das Schlüsselwort :TEST gefolgt von 'EQUAL an. Wir haben dann

```
-->(MEMBER '(A B) '((A B) (C D)) :TEST 'EQUAL)
((A B) (C D))
```

Das zu benutzende Testprädikat muß nicht einmal eine Systemfunktion sein; es kann auch eine mit den Mitteln der nächsten Lektionen selbstdefinierte Funktion sein.

Die Funktion MEMBER ist kein Prädikat im eigentlichen Sinn, da der Wertebereich nicht nur aus den beiden Werten T und NIL besteht. Sie kann aber wie ein Prädikat benutzt werden, weil jeder Wert ungleich NIL in Positionen, in denen ein Wahrheitswert erwartet wird, als Ersatz für T behandelt wird. Man beachte jedoch

```
-->(EQ T (MEMBER 'Y '(X Y Z)))
NIL
```

und ebenso

```
-->(EQUAL T (MEMBER 'Y '(X Y Z)))
NIL
```

4.6 Zusammenfassung

Zum Überprüfen von Bedingungen braucht man Prädikate.
Zu den wichtigsten gehören ATOM , CONSP und LISTP .

Sie überprüfen, ob der Wert eines Ausdruck

- ein Atom ist, bzw.
- mit CONS aufgebaut werden kann, bzw.
- eine Liste ist, d.h. gleich NIL ist oder mit CONS aufgebaut werden kann.

Die Übereinstimmung von LISP-Strukturen kann mit EQ , EQUAL oder EQL getestet werden:

EQ testet die Identität der internen Darstellung von Ausdrücken.

EQUAL testet die Übereinstimmung der externen Darstellung von Ausdrücken und mit Einschränkungen die Gleichheit von Zahlen.

EQL testet die Identität der internen Darstellung von Ausdrücken wie EQ und mit denselben Einschränkungen wie EQUAL die Gleichheit von Zahlen.

Zahlen verschiedenen Typs werden mit = auf Wertgleichheit getestet .
Ungleichheit von Zahlen testet man mit /= ; für Größenvergleiche dienen < , <= , > und >=.
Mit Hilfe der Funktion MEMBER kann festgestellt werden, ob ein Ausdruck in einer Liste auftritt.
MEMBER ist strenggenommen kein Prädikat, da es im positiven Fall mehr Information liefert als nur das Symbol T.

KAPITEL 5:

Bedingungen

In diesem Kapitel erweitern wir zunächst unser Repertoire an Prädikaten und gehen anschließend auf logische Funktionen zur Verknüpfung von Prädikaten ein.
Dann lernen wir die Systemfunktion COND (von condition = Bedingung) kennen. Sie ermöglicht die Verwendung von Bedingungen zur Steuerung des Programmablaufs.
Zum Abschluß dieses Kapitels werden noch einige Funktionen für Operationen auf Listen eingeführt.

5.1 Noch einige Prädikate

Wir erwähnen hier einige Prädikate, die häufig benutzt werden und deshalb auch als Systemfunktionen zur Verfügung stehen.

Das Prädikat NULL testet sein Argument, ob es die leere Liste darstellt.

Aufrufmuster:
(NULL <Ausdruck>)

Wert:
T falls das Argument als Wert die leere Liste hat
NIL sonst.

Seiteneffekt: Keiner

Das Prädikat NUMBERP testet sein Argument, ob es eine Zahl darstellt.

Aufrufmuster:
(NUMBERP <Ausdruck>)

Wert:
T falls das Argument als Wert eine Zahl liefert
NIL sonst.

Seiteneffekt: Keiner

Das Prädikat ZEROP testet eine Zahl, ob sie den Wert 0 hat.

Aufrufmuster:
(ZEROP <-Zahl->)

Wert:
T falls das Argument eine Zahl mit dem Wert 0 darstellt.
NIL falls das Argument eine Zahl mit einem Wert ungleich 0 darstellt.

Eine Fehlermeldung wird ausgegeben, falls das Argument keine Zahl darstellt.

Seiteneffekt: Keiner

Beispiele:

```
-->(NULL '( ))
T

-->(NULL NIL)
T

-->(SETQ L '( ))
NIL

-->(NULL L)
T

-->(NULL (CDDR '(A B)))
T

-->(NULL 0)
NIL

-->(NUMBERP '3.6)
T
```

```
-->(NUMBERP 'X)
NIL

-->(ZEROP 0)
T

-->(ZEROP '0)
T

-->(ZEROP '3)
NIL

-->(ZEROP 'X)
ERROR                                   ;  ZEROP ist nur für Zahlen definiert
```

Das Prädikat SYMBOLP testet sein Argument, ob es ein Symbol darstellt.

Aufrufmuster:
(SYMBOLP <Ausdruck>)

Wert:
T falls das Argument ein Symbol als Wert hat
NIL sonst.

Seiteneffekt: Keiner

Das Prädikat STRINGP testet sein Argument, ob es einen String darstellt.

Aufrufmuster:
(STRINGP <Ausdruck>)

Wert:
T falls das Argument einen String als Wert hat
NIL sonst.

Seiteneffekt: Keiner

Beispiele:

```
-->(SYMBOLP 3.0)
NIL

-->(STRINGP 'AB)
NIL

-->(SYMBOLP 'AB)
T

-->(STRING "AO")
T
```

Viele weitere Prädikate stehen ebenfalls als Systemfunktionen zur Verfügung. Wir werden sie jeweils im Bedarfsfalle einführen.

Noch eine Bemerkung zu den Bezeichnungen für Prädikate:
Häufig werden zur Bezeichnung von Prädikaten Namen verwendet, die mit dem Buchstaben P (für predicate) enden. Beispiele sind LISTP, NUMBERP und ZEROP. Dies erleichtert das Erkennen von Prädikaten und ist deshalb auch bei der Definition neuer (eigener) Prädikate sinnvoll. Die Definition von Funktionen und Prädikaten durch den Benutzer werden wir in dem nächsten Kapitel kennenlernen.

5.2 Typprüfungen

Wir kennen inzwischen einige Typen von LISP-Objekten, nämlich Atome, Cons-Paare und Listen, sowie Zahlen, Symbole und Strings.

Mit Hilfe der Prädikate ATOM, CONSP, LIST, NUMBERP, SYMBOLP und STRINGP können wir feststellen, ob ein beliebiger Ausdruck ein Atom, ein Cons-Paar usw. darstellt.

COMMON LISP kennt aber nicht nur die erwähnten einfachen Datentypen, sondern viele weitere Datentypen stehen standardmäßig zur Verfügung; darüber hinaus kann auch der Benutzer selbst Datentypen definieren.
Zu jedem Datentyp gehört ein Typ-Spezifikator, der den Datentyp beschreibt. Die oben genannten einfachen Datentypen haben je ein Symbol als Spezifikator und zwar der Reihe nach ATOM,

CONS, LIST, NUMBER, SYMBOL und STRING. Diese Spezifikatoren sind keine Prädikate, können aber als Argumente bei Typprüfungen mit dem Prädikat TYPEP verwendet werden. TYPEP hat das
Aufrufmuster:

(TYPEP <LISP-Objekt> <Typ-Spezifikator>)

Beispiele:
-->(NUMBERP 3.0)
T
-->(TYPEP 3.0 NUMBER)
T
-->(TYPEP 3.0 SYMBOL)
NIL

Verschiedene Datentypen sind nicht notwendigerweise disjunkt. Insbesondere kann ein Typ auch Subtypen haben. So hat der Typ ATOM die Subtypen SYMBOL, NUMBER und STRING. Der Typ NUMBER hat viele Subtypen, unter anderem INTEGER mit den Subtypen FIXNUM und BIGNUM, dann SINGLE-FLOAT, DOUBLE-FLOAT, LONG-FLOAT, RATIONAL und COMPLEX für ganze Zahlen verschiedener Länge, Gleitpunktzahlen verschiedener Genauigkeit, rationale und komplexe Zahlen.

Wir wollen die Betrachtungen über Zahlentypen nicht vertiefen, da uns bei LISP vor allem die Möglichkeiten des symbolischen Rechnens, weniger die Durchführung numerischer Berechnungen interessieren.

Beispiele:
-->(TYPEP -3.06 INTEGER)
NIL
-->(TYPEP -3.06 SINGLE-FLOAT)
T

5.3 Die logischen Verknüpfungen AND, OR und NOT

Um komplexere Prädikate formulieren zu können, benötigen wir logische Verknüpfungen für die Wahrheitswerte. Dazu dienen die Systemfunktionen AND, OR und NOT mit den

Aufrufmustern:

(AND <Ausdruck-1> <Ausdruck-2> ... <Ausdruck-n>) mit n ≥ 0

(OR <Ausdruck-1> <Ausdruck-2> ... <Ausdruck-n>) mit n ≥ 0

(NOT <Ausdruck>)

Zur Auswertung von AND:

AND evaluiert im Falle n ≥ 1 die Argumente der Reihe nach von links nach rechts: Tritt dabei der Wert NIL auf, so wird sofort NIL als Wert der AND-Form ausgegeben und auf die Auswertung der restlichen Argumente verzichtet. Letzteres ist dann von Bedeutung, wenn die Auswertung der restlichen Argumente Seiteneffekte hätte.
Hat keines der Argumente den Wert NIL, so erhält die AND-Form den Wert des letzten Arguments.
Im Falle n = 0 liefert die Auswertung T.

Beispiele:

```
-->(AND (NUMBERP 3) (SETQ ACHTUNG 'SEITENEFFEKT) (NULL 'T))
NIL

-->(AND (NUMBERP 3) (NULL 'T) (SETQ ACHTUNG 'SEITENEFFEKT))
NIL
```

Beide AND-Formen liefern den Wert NIL. Die Auswertung der ersten hat aber einen Seiteneffekt, die der zweiten nicht. Ursache hierfür ist, daß die Auswertung der AND-Form abgebrochen wird, sobald ein Argument angetroffen wird, das den Wert NIL hat.

```
-->(AND)
T

-->(AND (NULL '( )) (MEMBER 'X3 '(X1 X2 X3 X4 X5)))
(X3 X4 X5)                          ; dies ist der Wert des letzten Arguments
```

Zur Auswertung von OR:

Die Funktion OR evaluiert im Falle n ≥ 1 die Argumente ebenfalls der Reihe nach von links nach rechts. Tritt dabei ein Wert ungleich NIL auf, so wird dieser als Wert der OR-Form ausgegeben und auf die Auswertung der restlichen Argumente verzichtet. Ergeben alle Argumente den Wert NIL, so wird NIL auch Wert der OR-Form.
Im Falle n = 0 liefert die Auswertung den Wert NIL.

Beispiele:

```
-->(OR (MEMBER 'X6 '(X1 X2 X3 X4 X5)) (NULL '0))
NIL

-->(OR (MEMBER 'X3 '(X1 X2 X3 X4 X5)) (NULL 'NIL))
(X3 X4 X5)

-->(OR (NULL 'NIL) (MEMBER 'X3 '(X1 X2 X3 X4 X5)))
T
```

Die beiden OR-Formen liefern unterschiedliche Werte, weil die Auswertung jeweils nach dem ersten Wert ungleich NIL abgebrochen wird.

```
-->(OR)
NIL
```

Zur Auswertung von NOT:

Die Funktion NOT liefert den Wert NIL, falls sein Argument einen Wert ungleich NIL hat; anderenfalls den Wert T.

Beispiele:

```
-->(NOT 'NIL)
T

-->(NOT 'X)
NIL                    ; dasselbe Resultat ergibt sich, wenn man
                       ; anstatt 'X einen beliebigen Ausdruck mit
                       ; Wert ungleich NIL einsetzt
```

AND, OR und NOT haben selbst keine Seiteneffekte; natürlich kann die Auswertung der Argumente Seiteneffekte bewirken.

AND und OR sind aufgrund der besonderen Regeln zur Auswertung ihrer Argumente spezielle Formen.

Bemerkung:
Aufgrund der Äquivalenz von NIL und der leeren Liste sind die Systemfunktionen NOT und NULL äquivalent, d.h. die Werte von

(NOT X) und (NULL X)

stimmen für jedes Argument überein und beide haben denselben Seiteneffekt, nämlich keinen.

5.4 Bedingungen, die spezielle Form COND

Die spezielle Form COND (von condition = Bedingung) gibt uns die Möglichkeit, Ausdrücke zu formulieren, deren Werte von Bedingungen abhängen.

Aufrufmuster:

```
(COND     (<Test-1> <Programmstück-1>)
          (<Test-2> <Programmstück-2>)
              .
              .
              .
              .
          (<Test-n> <Programmstück-n>))        mit n ≥ 1
```

Unter einem Programmstück verstehen wir hier eine Folge von null, einer oder mehr Formen.

Die auf das Symbol COND folgenden Listen der Gestalt

(<Test> <Programmstück>)

nennt man Klauseln.

Auswertung:
Die jeweils ersten Formen der Klauseln heißen Tests. Sie werden in der vorgegebenen Reihenfolge ausgewertet - einer nach dem anderen - und zwar solange, bis ein Test einen Wert ungleich NIL ergibt. Zu diesem positiven Test wird dann auch der Rest der zugehörigen Klausel ausgewertet. Der Wert der letzten Form dieser Klausel wird der Wert der COND-Form. Die restlichen Klauseln werden nicht mehr ausgewertet.
Ergibt kein Test einen Wert ungleich NIL, so hat die COND-Form den Wert NIL.

Es ist möglich, daß eine Klausel nur aus einer Form, nämlich dem Test besteht; das zugehörige Programmstück ist dann leer. In diesem Fall wird der Wert des positiven Tests auch der Wert der COND-Form.

Seiteneffekt: Außer den Seiteneffekten, die mit den ausgewerteten Tests und dem zum positiven Test ausgewerteten Programmstück zusammenhängen, keine weiteren Seiteneffekte.

Beispiele:

```
(COND (X Y) (T NIL))          ; hat denselben Wert wie (AND X Y)
(COND (X X) (T Y))            ; hat denselben Wert wie (OR X Y)
(COND ((EQ X NIL) T))         ; hat denselben Wert wie (NOT X)
```

```
(COND ((MEMBER ELEMENT LISTE) LISTE)
      ( T (CONS ELEMENT LISTE )))
                ; Wert dieser COND-Form ist
                ; der Wert von Liste, falls der Wert des Arguments
                ;           ELEMENT als Element in der Liste auf-
                ;           tritt, die Wert des Arguments LISTE ist;
                ; andernfalls der Wert von (CONS ELEMENT LISTE)
```

In jedem der obigen Beispielen ist die Testform der letzten Klausel gleich T. Dies ist zwar nicht vorgeschrieben, entspricht aber einer üblichen, sehr vernünftigen Praxis bei der Verwendung von COND. Falls keine der vorangehenden Testformen einen Wert ungleich NIL liefert, bewirkt die letzte Testform T, daß ihr Resultat als Wert der COND-Form übergeben wird. Auf diese Weise wird vermieden, daß die COND-Form einen Wert annimmt, der möglicherweise vom Programmierer nicht vorgesehen war.

Weiteres Beispiel:

```
(COND ((NULL L) NIL)
      ((EQUAL ELEMENT (CAR L)) L)
      ( T (MEMBER ELEMENT (CDR L))))
                ; als Wert ergibt sich
                ; NIL,  falls L die leere Liste ist
                ; L,    falls ELEMENT  im Wert mit dem ersten Element von L
                ;       übereinstimmt
                ; der Wert von  (MEMBER ELEMENT (CDR L))
                ;       sonst
```

Die obige COND-Form liefert für "Argumente" ELEMENT und L stets denselben Wert wie der Funktionsaufruf (MEMBER ELEMENT L).
Man könnte versucht sein, diese COND-Form als Definition für die Funktion MEMBER anzusehen. Dem scheint allerdings noch entgegenzustehen, daß die Funktion MEMBER in der COND-Form selbst aufgerufen wird.
Außerdem wissen wir noch nicht, wie Funktionen definiert werden können. Das werden wir jedoch in dem nächsten Kapitel erfahren. In dem übernächsten Kapitel werden wir zudem die Definition sogenannter rekursiver Funktionen kennenlernen und dann feststellen, daß obige COND-Form schnell zu einer Definition der Funktion MEMBER ausgebaut werden kann.

5.5 Weitere Funktionen zur Manipulation von Listen

Unser Repertoire an Funktionen für den Umgang mit Listen soll im folgenden ergänzt und abgerundet werden.

Die Funktion LENGTH liefert bei einem Aufruf der Gestalt

(LENGTH <-Liste->)

also für ein Argument, dessen Wert eine Liste ist, die Anzahl der Elemente dieser Liste und zwar auf dem obersten Niveau dieser Liste.

Die Funktion REVERSE liefert für einen Aufruf der Gestalt

(REVERSE <-Liste->)

also für ein Argument, dessen Wert eine Liste ist, eine Liste, die aus der gegebenen Liste hervorgeht, indem man auf dem obersten Niveau die Reihenfolge der Elemente invertiert; die Listenelemente selbst bleiben unangetastet;

Die Funktion LAST liefert für einen Aufruf der Gestalt

(LAST <-Liste->)

wieder mit einem Argument, dessen Wert eine Liste ist, eine Liste, die das letzte Elemente der gegebenen Liste als einziges Element enthält.

Die Funktionen **LENGTH , REVERSE** und **LAST** haben außer den mit der Auswertung des Arguments gegebenenfalls verknüpften Seiteneffekten keine Seiteneffekte.

Beispiele:

```
-->(SETQ L '(A B))
(A B)

-->(LENGTH L)
2

-->(LENGTH '(A B))
2

-->(LENGTH '((A B) (C D)))
2


-->(LENGTH (APPEND L '(A B)))
4

-->(LENGTH (LIST L '(A B)))
2

-->(LENGTH (MEMBER 'X3 '(X1 X2 X3 X4 X5)))
3

-->(REVERSE '(A B))
(B A)

-->(REVERSE L)
(B A)                          ; L wird im Wert nicht verändert

-->(REVERSE '((A B) (C D)))    ; man beachte, daß nur das oberste
((C D) (A B))                  ; Niveau bearbeitet wird
```

```
-->(REVERSE (APPEND L '(C D)))
(D C B A)

-->(LAST L)
(B)

-->(LAST '((A B) (C D)))
((C D))

-->(COND ((MEMBER (LAST '((A B) (C D))) '(A B) (C D)))
                    'FALSCH)
         ((< (LENGTH (APPEND '(L) L)) 4)
                    (REVERSE '(G I T H C I R))))
(R I C H T I G)
```

5.6 Zusammenfassung

Mit Prädikaten kann man überprüfen, ob eine Bedingung erfüllt ist oder nicht.
Die Systemfunktion COND gibt uns die Möglichkeit, für unterschiedliche Bedingungen jeweils ein bestimmtes Verhalten des Programmablaufs vorzuschreiben. Dazu werden vorgegebene Bedingungen in vorgegebener Reihenfolge überprüft. Die erste erfüllte Bedingung stellt die Weiche zur Auswertung des ihr zugeordneten Programmstücks.

KAPITEL 6:

Funktionen

Alle Funktionen, die wir bisher kennengelernt haben, sind Systemfunktionen, d.h. sie stehen in LISP-Systemen standardmäßig zur Verfügung. Im folgenden erfahren wir, wie der Benutzer selbst Funktionen definieren kann. In LISP besteht die Möglichkeit, benannte Funktionen und unbenannte (anonyme) Funktionen zu definieren. Dieses Kapitel beschäftigt sich mit ihrer Definition und ihrer Anwendung.

6.1 Definition benannter Funktionen

Ein frei gewähltes Symbol kann als Bezeichnung für eine Rechenvorschrift eingeführt werden. Dazu dient die Systemfunktion DEFUN (von define function) mit dem
Aufrufmuster:

(DEFUN <Funktionsname> <Parameterliste> <Funktionsrumpf>)

Der Funktionsname ist ein Symbol; die Elemente der Parameterliste sind Symbole, die im Funktionsrumpf auftreten können (nicht müssen !); der Funktionsrumpf ist ein Programmstück, bestehend aus einer Folge von null, einer oder mehreren Formen.

Die **Auswertung** einer Funktionsdefinition mit DEFUN liefert als
Wert den Funktionsnamen - das interessiert hier weniger - und bewirkt als
Seiteneffekt eine Zuordnung des Funktionsnamens an den Funktionsrumpf.

Diese Zuordnung hat zur Folge, daß der Funktionsrumpf ausgewertet wird, wenn der Funktionsname als erstes Listenelement in einem auszuwertenden Ausdruck auftritt. Wir sprechen dabei auch von einem **Funktionsaufruf**. Er hat wie bei Systemfunktionen die Gestalt:

(<Funktionsname> <Argument-1> <Argument-2> ... <Argument-n>)

Die Anzahl n der benötigten Argumente ist durch die Anzahl der Parameter in der Parameterliste der zugrundeliegenden Funktionsdefinition festgelegt. Argument-1 bis Argument-n sind Ausdrücke.

Zur Auswertung des Funktionsaufrufs werden zuerst die Argumente Argument-1 bis Argument-n in dieser Reihenfolge ausgewertet. Die Werte dieser Argumente werden dann der Reihe nach den Parametern aus der Parameterliste als Anfangswerte zugeordnet. Unter Beachtung dieser Zuordnungen wird sodann der Funktionsrumpf ausgewertet, d.h. seine Formen werden der Reihe nach ausgewertet. Der Wert der letzten Form im Funktionsrumpf wird Wert des Funktionsaufrufs. Wenn der Funktionsrumpf aus null Formen besteht, d.h. leer ist, so ist der Wert des Funktionsaufrufs gleich NIL.
Sobald die Funktionsauswertung abgeschlossen ist, werden alle Wertzuordnungen an die Parameter wieder aufgehoben.

Beispiel:

```
-->(DEFUN VERTAUSCHE (PAAR)                ; Funktionsdefinition
     (LIST (CADR PAAR) (CAR PAAR)))
VERTAUSCHE                                 ; Funktionsname als Wert

-->(SETQ ZWILLINGE '(MAX MORITZ))
(MAX MORITZ)

-->(VERTAUSCHE ZWILLINGE)                  ; Funktionsaufruf
(MORITZ MAX)
```

Wie wir festgestellt haben, wird unmittelbar nach Abschluß der Funktionsauswertung jede Zuordnung von Werten an die Funktionsparameter wieder aufgehoben.

Eine Zuordnung, deren Gültigkeit auf den textuellen Bereich eines Sprachkonstrukts (hier des Funktionsrumpfes) beschänkt ist und mit dem Verlassen des Sprachkonstrukts wieder aufgehoben wird, nennen wir eine **Bindung**. Wir sagen deshalb auch, die Parameter der Funktion werden bei Beginn der Auswertung an die Werte der Argumente **gebunden**.

Für den Namen einer bereits definierten Funktion, kann durch einen erneuten DEFUN-Aufruf eine neue Funktionsdefinition vorgenommen werden. Auf diese Weise lassen sich sogar Systemfunktionen überschreiben.

Übrigens ist DEFUN ein weiteres Beispiel für eine spezielle Form im Sinne von Kapitel 3; zu ihrer Auswertung wird keines ihrer Argumente ausgewertet.

6.2 Definition unbenannter Funktionen, Lambda-Ausdrücke

Im folgenden behandeln wir die Definition von namenlosen (oder anonymen) Funktionen durch eine Rechenvorschrift.
Dazu dienen sogenannte **Lambda-Ausdrücke** der Gestalt

(LAMBDA <Parameterliste> <Funktionsrumpf>)

> Die Parameterliste und der Funktionsrumpf haben dieselbe Bedeutung wie bei mit DEFUN definierten Funktionen: Die Parameterliste enthält Symbole, die im Funktionsrumpf auftreten können (nicht müssen !). Der Funktionsrumpf ist ein Programmstück, bestehend aus einer beliebigen Anzahl von Formen.

Ein solcher Lambda-Ausdruck beschreibt eine namenslose Funktion; er ist für sich gesehen keine auswertbare Form, d.h. für Lambda-Ausdrücke allein ist in LISP kein Wert definiert. Ein auswertbarer Ausdruck entsteht erst, wenn der Lambda-Ausdruck mit Argumenten zu einem **Funktionsaufruf** zusammengefaßt wird. Dieser hat dann die Gestalt

(<Lambda-Ausdruck> <Argument-1> <Argument-2> ... <Argument-n>)

Die Auswertung erfolgt genauso, wie oben im Falle der mit DEFUN definierten Funktionen beschrieben.

Beispiel:
Der Lambda-Ausdruck

(LAMBDA (PAAR) (LIST (CADR PAAR) (CAR PAAR)))

beschreibt die Vertauschung der Elemente einer zweielementigen Liste, ist aber kein auswertbarer Ausdruck. Durch Angabe von Argumenten für die Parameter entsteht ein auswertbarer Funktionsaufruf:

```
-->((LAMBDA (PAAR) (LIST (CADR PAAR) (CAR PAAR))) '(MANN FRAU))
(FRAU MANN)
```

Der Unterschied zwischen der Definition einer Funktion durch eine DEFUN-Form und der Definition durch einen Lambda-Ausdruck besteht allein darin, daß im ersten Fall ein Name für die Rechenvorschrift eingeführt wird, im zweiten Fall dagegen nicht. Die Auswertung einer DEFUN-Form bewirkt gerade die Zuordnung des Funktionsnamens an den mit Parameterliste und Funktionsrumpf gebildeten Lambda-Ausdruck. Hinsichtlich der Argumentübergabe und der Abarbeitung des Funktionsrumpfes bestehen keine Unterschiede. Die Zuordnung eines Namens an eine Funktion macht es möglich, daß die Definition der Funktion und ihre Verwendung unter dem eingeführten Namen beliebig weit auseinanderliegen können. Ist eine Funktion aber ohne Namen, also nur mit einem Lambda-Ausdruck definiert, so kann sie nur an der Stelle ihrer Definition ausgewertet werden.

Die Verwendung des Symbols LAMBDA und die Bezeichnung Lambda-Ausdruck sind historisch bedingt. Das Grundkonzept von LISP basiert nämlich auf dem sogenannten Lambda-Kalkül von Church (siehe Church 1941). Man könnte sich informativere Symbole als LAMBDA vorstellen, etwa ANFUN (für anonyme Funktion) als Gegensatz zu DEFUN.
Die Auswertung eines Funktionsaufrufes heißt im Falle der benannten Funktion wie im Falle eines unbenannten Lambda-Ausdrucks auch heute noch Lambda-Konversion.

Grundsätzlich dürfen in LISP in allen Positionen, in denen Funktionsnamen auftreten können, auch Lambda-Ausdrücke verwendet werden. Ein Funktionsaufruf hat damit allgemein die Gestalt

(<Funktionsbeschreibung> <Argument-1> ... <Argument-n>)

wobei die Funktionsbeschreibung ein Funktionsname oder aber ein Lambda-Ausdruck sein kann.

6.3 Beispiele zur Definition von Funktionen

Ein erstes Beispiel
zeigt die Definition einer Funktion OUR-ADJOIN, welche die Hinzunahme eines Objektes zu einer Liste veranlaßt, falls das Objekt noch nicht Element der Liste ist.

```
(DEFUN OUR-ADJOIN (OBJEKT LISTE)
   (COND ((MEMBER OBJEKT LISTE) LISTE)
         (T (CONS OBJEKT LISTE))))
```

Hier wird zunächst geprüft, ob der Wert von OBJEKT schon Element von LISTE ist. Wenn ja, bleibt diese unverändert; anderenfalls wird der Wert von OBJEKT als neues erstes Element in LISTE aufgenommen.

Wir geben noch den äquivalenten Lambda-Ausdruck an:

```
(LAMBDA (OBJEKT LISTE)
   (COND ((MEMBER OBJEKT LISTE) LISTE)
         (T (CONS OBJEKT LISTE))))
```

Das zweite Beispiel baut auf dem ersten auf.
Angenommen, wir wollten OUR-ADJOIN mit 2 Argumenten aufrufen, haben aber vergessen, welches dieser Argumente das Objekt und welches die Liste darstellt. Wir schreiben deshalb eine neue Version, genannt AO-ADJOIN (mit AO für any order), das die Argumente in beliebiger Reihenfolge akzeptiert und selbst testet, welches der Argumente die Liste darstellt. Dabei setzen wir voraus, daß nur eines der Argumente eine Liste ist.

Wir könnten einfach

```
          (OUR-ADJOIN ZWEITES ERSTES)
```

und `(OUR-ADJOIN ERSTES ZWEITES)`
aufrufen und überprüfen, wo die Fehlermeldung erfolgt.
Das Risiko, daß die Berechnung aufgrund eines Fehlers unterbrochen wird, können wir durch Einbetten in eine COND-Form mit entsprechenden Abfragen vermeiden. Eine derartige COND-Form ist gegeben durch

```
(COND   ((LISTP ERSTES)
           (OUR-ADJOIN ZWEITES ERSTES))
        (T  (OUR-ADJOIN ERSTES ZWEITES)))
```

Wir vervollständigen sie noch zu einer Funktionsdefinition für eine Funktion Any Order-ADJOIN:

```
(DEFUN AO-ADJOIN (ERSTES ZWEITES)
   (COND ((LISTP ERSTES)
             (OUR-ADJOIN ZWEITES ERSTES))
         (T    (OUR-ADJOIN ERSTES ZWEITES))))
```

Eleganter ist folgende Lösung mit nur einem Aufruf von OUR-ADJOIN:

```
(SETQ   OBJEKT ERSTES
        LISTE ZWEITES)
(COND ((LISTP ERSTES)
          (SETQ  OBJEKT ZWEITES
                 LISTE ERSTES)))
(OUR-ADJOIN OBJEKT LISTE)
```

Durch die COND-Form werden jetzt die Argumente für OUR-ADJOIN im Falle falscher Anordnung einfach ausgetauscht.

Hierfür kann man aber auch einen Lambda-Ausdruck definieren, und die erste versuchsweise Zuordnung von OBJEKT an ERSTES sowie LISTE an ZWEITES durch Parameterübergabe erfassen.

```
((LAMBDA (OBJEKT LISTE)
   (COND ((LISTP ERSTES)
             (SETQ  OBJEKT ZWEITES
                    LISTE ERSTES)))
   (OUR-ADJOIN  OBJEKT LISTE))
                                    ERSTES ZWEITES)
```

Dieser Lambda-Ausdruck wird mit den Argumenten ERSTES und ZWEITES aufgerufen. Bei seiner Auswertung wir die Bindung der Parameter zunächst versuchsweise vorgenommen und durch die COND-Form im Falle falscher Anordnung korrigiert.

Ein flexiblerer Einsatz wird möglich, wenn diese Lambda-Anwendung als Rumpf einer Funktionsdefinition mit DEFUN verwendet wird. Wir kommen damit zu folgender Definition für eine Funktion Smart-Any-Order-ADJOIN:

```
(DEFUN SAO-ADJOIN (ERSTES ZWEITES)
   ((LAMBDA (OBJEKT LISTE)
       (COND ((LISTP ERSTES)
                 (SETQ  OBJEKT ZWEITES
                        LIST ERSTES)))
       (OUR-ADJOIN OBJEKT LISTE))
                 ERSTES ZWEITES))
```

Die Symbole ERSTES und ZWEITES sind formale Parameter von SAO-ADJOIN, aber Argumente im Aufruf des Lambda-Ausdrucks.

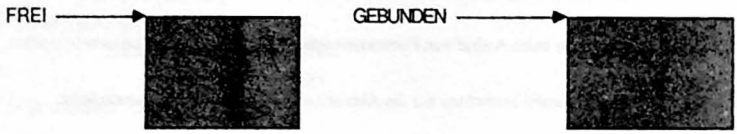
6.4 Argumentübergabe beim Funktionsaufruf

Die Argumentübergabe beim Aufruf von Funktionen soll im folgenden genauer untersucht werden.

Als Demonstrationsbeispiel betrachten wir die Abarbeitung des folgendes Programmstücks:

```
-->(SETQ FREI 3)
3

-->(DEFUN DEMO (GEBUNDEN)
     (SETQ GEBUNDEN (+ GEBUNDEN 10))
     (+ GEBUNDEN FREI))
DEMO

-->(DEMO FREI)
16

-->FREI
3
```

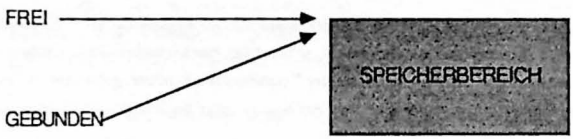
Bei jeder Auswertung einer Funktion werden zunächst die Argumente im Funktionsaufruf der Reihe nach von links nach rechts ausgewertet; an die dabei resultierenden Werte werden sodann die einzelnen Parameter aus der Parameterliste der Definition der Funktion gebunden und zwar in der vorgegebenen Reihenfolge; schließlich wird der Rumpf unter Berücksichtigung dieser Bindungen ausgewertet.
Die Parameterliste in der Definition der Funktion DEMO enthält genau einen Parameter, nämlich das Symbol GEBUNDEN. Im Aufruf von DEMO wird dementsprechend genau ein Argument, hier das Symbol FREI angegeben. Durch die Argumentübergabe wird der Parameter GEBUNDEN an den Wert 3 des Arguments FREI gebunden. Während der Auswertung des Funktionsrumpfes von DEMO wird der Wert von GEBUNDEN verändert. Dies beeinflußt den Wert von FREI nicht. Er bleibt während der ganzen Funktionsauswertung unverändert. Deshalb liefert der obige Aufruf (DEMO FREI) auch den Wert 16.
Die Bindung des Parameters an die Argumentwerte hat man sich wie folgt vorzustellen:
Bei Abarbeitung des Funktionsaufrufs (DEMO FREI) wird als erstes ein Speicherbereich für den Parameter GEBUNDEN angelegt und zwar unabhängig vom Speicherbereich für FREI. Sodann wird in den Speicherbereich für GEBUNDEN der Wert von FREI zum Zeitpunkt der Argumentauswertung eingetragen.



Jetzt können nachfolgende Wertänderungen bei GEBUNDEN den Wert von FREI nicht beeinflussen und umgekehrt.

Diese Art der Argumentübergabe (auch Parameterübergabe genannt) bezeichnet man kurz als Wertübergabe (auch Wertaufruf oder " Call by value").

Es gibt auch andere Möglichkeiten der Argumentübergabe, zum Beispiel mit Übergabe der Referenz (sogenannter Referenzaufruf oder "Call by reference"). Wenn LISP den Referenzaufruf realisieren würde, würde dem Parameter GEBUNDEN beim Funktionsaufruf derjenige Speicherbereich zugeordnet, der zum Zeitpunkt des Aufrufs dem Argument FREI zugeordnet ist.



Die Werte von FREI und GEBUNDEN wären in ein- und demselben Speicherbereich abgelegt; jede Änderung von FREI würde auch eine Änderung von GEBUNDEN bedeuten und umgekehrt.

6.5 Freie und gebundene Variable

Symbole, die verschiedene Werte annehmen können, nennt man üblicherweise **Variable**.
Eine Variable heißt bezüglich einer Funktion **gebunden** oder **lokal**, wenn sie in der Parameterliste der Funktionsdefinition auftritt; sie heißt bezüglich einer Funktion **frei**, wenn sie im Funktionsrumpf, aber nicht in der Parameterliste auftritt.

Beispiel:

Die im vorhergehenden Abschnitt definierte Demonstrationsfunktion DEMO mit der Definition

```
(DEFUN DEMO (GEBUNDEN)
   (SETQ GEBUNDEN (+ GEBUNDEN 10))
   (+ FREI GEBUNDEN))
```

enthält die Variablen FREI und GEBUNDEN; die Variable FREI ist frei, die Variable GEBUNDEN ist gebunden.

Jede Aussage, daß eine Variable frei oder gebunden ist, ist nur sinnvoll, wenn gleichzeitig gesagt wird oder klar ist, auf welche Funktion sich diese Aussage bezieht.

Die Variable FREI tritt im Rumpf von DEMO auf, wird aber dort nicht verändert. Nun modifizieren wir DEMO zu DAMO (von Damokles) wie folgt:

```
-->(SETQ FREI 3)
3

-->(DEFUN DAMO (GEBUNDEN)
        (SETQ GEBUNDEN (+ GEBUNDEN 10))
      (+ FREI (SETQ FREI GEBUNDEN)))          ; Wertzuweisung an FREI
DAMO

-->(DAMO FREI)
16
-->FREI                  ; Wert von FREI wurde
13                       ; auf Dauer verändert
```

Die Funktion DAMO hat einen **Seiteneffekt**: ihre Auswertung liefert nicht nur einen Wert, sondern verändert auch den Wert der freien Variablen FREI im Speicher.

Seiteneffekte von Funktionen sind häufig die Ursache, wenn das Verhalten eines Programms schwer verständlich ist, weil Variable verändert werden, aber die Ursachen nicht offensichtlich sind. Seiteneffekte von Funktionen sind deshalb häufig Quellen für schwer zu entdeckende Fehler und sollten nur unter größter Umsicht eingesetzt werden.

Freie Variablen heißen so, weil ihre Bedeutung außerhalb der Funktion (frei von dieser) festgelegt ist.
Die Bedeutung einer gebundenen (lokalen) Variablen ist ganz auf die Funktion beschränkt (an sie gebunden). Eine Umbenennung einer gebundenen Variablen in der Definition der ihr zugrundeliegenden Funktion zeigt keine Auswirkungen, solange keine Übereinstimmung mit der Bezeichnung freier Variabler auftritt.

Was geschieht aber, wenn eine gebundene Variable einer Funktion mit einer außerhalb der Funktion definierten Variablen in der Bezeichnung übereinstimmt ?
Da jede gebundene Variable für die Auswertung des Funktionsaufrufs an den Wert des übergebenen Arguments gebunden wird, muß in diesem Fall die zum Zeitpunkt des Funktionsaufrufs bestehende Zuordnung vorübergehend außer Kraft gesetzt werden. Sie wird aber nach Abschluß der Auswertung wieder hergestellt.

Beispiel:

```
-->(SETQ FREI 3)
3
-->(DEFUN DEMO (FREI)
      (SETQ FREI (+ FREI 10))
      (+ FREI FREI))
DEMO
-->(DEMO 2)                    ; Zu Beginn der Auswertung von DEMO
24                             ; wird die Bindung "FREI an 2" einge-
                               ; richtet, aber während der Auswertung
                               ; durch "FREI an 12" ersetzt
-->FREI                        ; Nach Abschluß der Auswertung von
3                              ; DEMO wird die ursprüngliche Zuordnung
                               ; von FREI an den Wert 3 restauriert
-->(DEMO (+ FREI FREI))        ; Hier wird die Bindung der gebundenen
32                             ; Variablen FREI mit Hilfe des Wertes
                               ; der freien Variablen FREI festgelegt:
                               ; Während der Abarbeitung des Rumpfes
                               ; bleibt die ursprüngliche Zuordnung von
                               ; FREI an 3 durch die eingerichtete Bindung
                               ; überdeckt
-->FREI                        ; Nach Abschluß der Auswertung des
3                              ; Funktionsaufrufs wird die ursprüngliche
                               ; Zuordnung von FREI restauriert.
```

6.6 Zusammenfassung

Gewisse Funktionen stehen an LISP-Systemen standardmäßig zur Verfügung; sie heißen Systemfunktionen.

Weitere Funktionen können wir in LISP-Programmen selbst definieren, entweder als benannte oder als unbenannte Funktionen. Bei der Definition einer benannten Funktion mit DEFUN wird einer Rechenvorschrift ein Symbol als Name zugeordnet. Diese Zuordnung wird durch Auswertung der DEFUN-Form eingerichtet. Die Rechenvorschrift kann dann an beliebiger Stelle im Programm unter diesem Namen mit geeigneten Argumenten aufgerufen werden. Bei Definition einer unbenannten Funktion in Form eines LAMBDA-Ausdrucks wird ebenfalls eine Rechenvorschrift formuliert; sie kann jedoch nur am Ort der Definition ausgewertet werden.
Die Übergabe der Argumente beim Aufruf einer Funktion erfolgt grundsätzlich nach den Regeln des Wertaufrufs.

Vorsicht ist immer geboten, wenn die Definition einer Funktion auch Variable enthält, die nicht in der Parameterliste auftreten und damit - wie man sagt - nicht gebunden, sondern frei sind.
Also Augen auf bei freien Variablen.

KAPITEL 7:

Rekursion

Als rekursiv bezeichnet man die Beschreibung eines Objektes (wie einer Datenstruktur oder einer Rechenvorschrift), wenn sie das Objekt unter Bezugnahme auf Objekte derselben Art beschreibt. Das klingt kompliziert. Tatsächlich ist Rekursion ein sehr natürliches Instrument, um die Lösung komplizierter Aufgaben auf die Lösung einfacherer und schließlich trivialer Aufgaben derselben Art zurückzuführen. Dieses Kapitel beschäftigt sich mit rekursiven Funktionen.

7.1 Grundprinzipien zur Definition rekursiver Funktionen

Wir beginnen mit einem einfachen **Beispiel:**
Angenommen, wir haben für einen Auftraggeber die Länge einer Liste L zu bestimmen.
Wir überprüfen zuerst, ob L leer ist:
Wenn L leer ist, melden wir unserem Auftraggeber die Zahl 0 als Länge von L.
Wenn L nicht leer ist, entfernen wir das erste Listenelement, reichen die Aufgabe, die Länge der verkürzten Liste zu bestimmen, an einen "Subunternehmer" weiter und warten auf dessen Meldung. Den uns von ihm schließlich gelieferten Wert erhöhen wir um 1 und melden diesen erhöhten Wert an unseren Auftraggeber.
Unser Subunternehmer verhält sich genauso wie wir.

Mit jeder Weitergabe der Aufgabe wird die Liste, deren Länge noch zu bestimmen ist, kürzer. Irgendwann wird nur noch die leere Liste weitergegeben. Deren Länge wird unmittelbar erkannt und zurückgemeldet. Es kommt zu einer Folge von Rückmeldungen. Jeder Subunternehmer meldet seinem Auftraggeber die Länge der erhaltenen Liste, dieser gibt den um 1 erhöhten Wert an seinen Auftraggeber weiter, bis der ursprüngliche Auftraggeber erreicht ist.
Hier wird also die Lösung der Aufgabe, die Länge einer Liste zu bestimmen, zurückgeführt auf die Lösung derselben Aufgabe, aber für eine kürzere Liste, und dies wird solange wiederholt, bis nur noch die triviale Aufgabe bleibt, die Länge der leeren Liste zu bestimmen.
Die Buchhaltung über die fortgesetzte Weitergabe der Aufgaben und die jeweilige Behandlung der Rückantwort erfordert zugleich beachtlichen Aufwand.

Nach dem beschriebenen Konzept definieren wir nun eine LISP-Funktion zur Bestimmung der Länge einer Liste L.

```
(DEFUN OUR-LENGTH (L)
    (COND ((NULL L) 0)
          (T (+ 1 (OUR-LENGTH (CDR L))))))
```

Der Funktionsrumpf ist eine COND-Form. In ihr wird zuerst geprüft, ob der einfache Fall vorliegt, in dem die Liste leer ist; dann ist nämlich die Länge gleich 0. Falls die Liste nicht leer ist, ergibt sich ihre Länge aus der Länge der um 1 Element verkürzten Liste durch Addition von 1. Die Länge der verkürzten Liste wird durch "rekursiven" Aufruf von OUR-LENGTH ermittelt. Diese Verkürzung und der nochmalige rekursive Aufruf von OUR-LENGTH für die verkürzte Liste wird solange fortgesetzt, bis OUR-LENGTH schließlich auf die leere Liste angewandt wird, also ein Fall vorliegt, in dem der Wert von OUR-LENGTH bekannt ist.

Allgemein heißt eine Funktion rekursiv, wenn sie sich selbst aufruft.
Korekursiv nennt man Funktionen, die sich gegenseitig aufrufen.

Abstrahieren wir von der speziellen Bedeutung der Funktion OUR-LENGTH im obigen Beispiel, so läßt sich folgendes Schema erkennen:

- Die COND-Form zur Beschreibung der Funktion beinhaltet eine Fallunterscheidung nach der Form des Arguments.

- Die einfachen Fälle, in denen der Funktionswert direkt angegeben werden kann, stehen am Anfang (im Beispiel ein einziger Fall, wenn nämlich die Liste leer ist).

- Darauf folgen diejenigen Fälle, in denen zur Berechnung des Funktionswertes die zu definierende Funktion selbst wieder (rekursiv) aufgerufen wird.

- Bei den inneren rekursiven Aufrufen werden die Argumente immer einfacher (im Beispiel wird die Liste immer kürzer), so daß nach endlich vielen Schritten ein Fall vorliegt, der so einfach ist, daß der Funktionswert direkt angegeben werden kann.

Diese Grundprinzipien finden wir bei jeder Definition einer rekursiven Funktion wieder. Dazu betrachten wir noch einige weitere Beispiele.

7.2 Einfache Beispiele für rekursive Funktionen

Beispiel:

Im folgenden definieren wir eine Funktion, die dasselbe leistet wie die Systemfunktion MEMBER, aber unter Zugrundelegung des Gleichheitstests mit EQUAL. Argumente sind ein Ausdruck und eine Liste. Falls der Ausdruck Element der Liste ist, soll diejenige Teilliste übergeben werden, welche mit dem ersten Auftreten des Ausdrucks beginnt, anderenfalls soll NIL als Resultat übergeben werden.

```
(DEFUN OUR-MEMBER (ELEMENT L)
    (COND ((NULL L) NIL)
          ((EQUAL ELEMENT (CAR L)) L)
          (T (OUR-MEMBER ELEMENT (CDR L)))))
```

Die Parameterliste enhält die Symbole ELEMENT und L zur Übernahme der Argumentwerte. Einfache Fälle, in denen der Funktionswert unmittelbar angegeben werden kann, haben wir hier zwei: Wenn die Liste leer ist, liefert die COND-Form das Resultat NIL; wenn das ELEMENT mit dem ersten Listenelement übereinstimmt, wird die Liste selbst übergeben. Liegt keiner dieser Fälle vor, so wird die Funktion rekursiv für die um das erste Element verkürzte Liste aufgerufen.

Beispiel:

Wir betrachten noch eine Funktion, welche die Anzahl des Auftretens von Atomen in einem Ausdruck L ermittelt:

```
(DEFUN COUNT-ATOMS (L)
    (COND ((NULL L) 0)
          ((ATOM L) 1)
            (T (+ (COUNT-ATOMS (CAR L)) (COUNT-ATOMS (CDR L))))))
```

Es wird jedes Auftreten eines Atoms in L mitgezählt. Wenn L leer ist oder ein Atom ist, dann ist der Funktionswert unmittelbar durch 0 bzw. 1 gegeben. Liegt keiner dieser einfachen Fälle vor, so wird L durch CAR und CDR aufgebrochen, dann werden rekursiv die Atome in den Teilstücken gezählt und die Anzahlen addiert.

Wir haben jetzt insgesamt drei Aufgaben gesehen, die mit Hilfe von rekursiven Funktionen sehr elegant gelöst werden können. Die Einfachheit und Kürze dieser rekursiven Lösungen wirkt überraschend, hat aber recht naheliegende Gründe.
In allen drei Fällen wird die Lösung der Aufgabe für die gegebene Liste zurückgeführt auf die Lösung derselben Aufgabe für die "natürlichen" Bestandteile der Liste, nämlich für ihre CAR- und CDR-Teile.
Aus diesen Bestandteilen sind Listen aber selbst rekursiv aufgebaut:

Eine Liste ist	entweder	die leere Liste
	oder entsteht	aus einer Liste durch Anfügen ("An-CONS-en") eines Elements.

Daß die Lösung der betrachteten Aufgaben mit Hilfe rekursiver Funktionen so einfach und elegant erscheint, liegt also ganz wesentlich daran, daß sich die Berechnung so eng an dem rekursiven Aufbau der zugrundeliegenden Listenstruktur orientieren konnte. Da LISP die Definition und Verwendung rekursiver Datenstrukturen besonders unterstützt, werden wir dieser Korrespondenz zwischen rekursiv aufgebauten Datenstrukturen und rekursiven Funktionen noch häufiger begegnen.

Beispiel:
Zur Übung wollen wir hier noch die im Kapitel 3 eingeführte Systemfunktion APPEND explizit definieren. Sie faßt mehrere Listen zu einer einzigen zusammen, beispielsweise die Listen (A B C) und (D E F) zu (A B C D E F).

Wir betrachten die Definition von APPEND für 2 Argumente:

Die Beobachtung, daß
(APPEND '(A B C) '(D E F)) und
(CONS (CAR '(A B C)) (APPEND '(B C) 'D E F)))
dasselbe liefern müssen, führt zu folgender rekursiver Definition einer Funktion OUR-APPEND für 2 Argumente:

```
(DEFUN  OUR-APPEND (X Y)
  (COND  ((NULL X) Y)
         (T (CONS (CAR X) (OUR-APPEND (CDR X) Y)))))
```

Beispiel:

Im folgenden betrachten wir eine rekursive Funktion, die über dem Bereich der natürlichen Zahlen definiert ist.

Sie geht zurück auf den italienischen Mathematiker Fibonacci und beschreibt die Entwicklung einer zunächst nur aus einem Paar bestehenden Kaninchenkolonie ohne Geburtenkontrolle und Tod.

Diese sogenannte Fibonacci-Funktion ist wie folgt rekursiv definiert:

$$f(n) = \begin{cases} 1 & \text{falls } n = 0 \text{ oder } n = 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Sie kann durch folgende LISP-Funktion beschrieben werden:

```
(DEFUN FIBONACCI (N)
  (COND ((= N 0) 1)
        ((= N 1) 1)
        (T (+ (FIBONACCI (- N 1)) (FIBONACCI (- N 2))))))
```

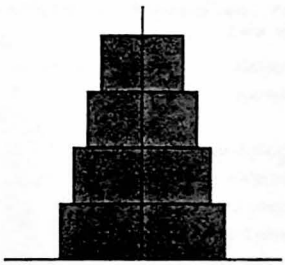
Wir haben zwei einfache Fälle, nämlich für N = 0 und N = 1. Für N > 1 wird (FIBONACCI N) auf zwei rekursive Aufrufe mit den Argumenten N-1 und N-2 zurückgeführt.

Es ist offensichtlich, daß die Auswertung anhand dieser rekursiven Definition nicht sehr effizient sein kann, da identische Probleme in lawinenartig anwachsender Zahl bearbeitet werden müssen.

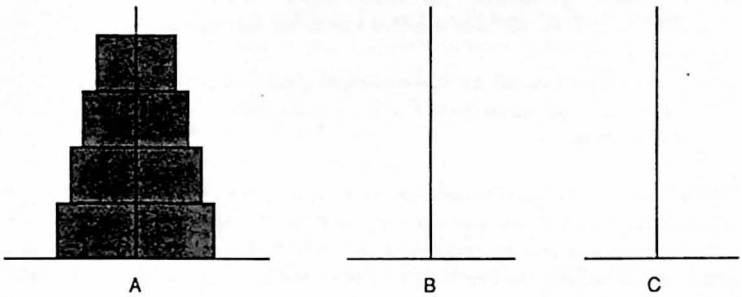
Rekursion und ihre Probleme waren schon im Altertum bekannt. Die Sage berichtet von den Großtaten des Herakles: Ihm ward auferlegt, gegen die 12-köpfige feuerspeiende Hydra zu kämpfen. Seine Versuche, die Bekämpfung der vielköpfigen Hydra durch Abschlagen eines Kopfes auf die leichtere Bekämpfung einer viel-minus-eins-köpfigen Hydra zurückzuführen, schlugen zunächst fehl: Für jeden abgetrennten Kopf wuchsen zwei neue nach. Nachdem sich die Köpfe schon fast auf die doppelte Anzahl vermehrt hatten und ihn von allen Seiten wütend angriffen, stellt er fest, daß er durch sofortiges Ausbrennen der Stümpfe das Nachwachsen von Köpfen verhindern konnte. So kam es doch noch zu einem Abbruch der Rekursion.

7.3 Die Türme von Hanoi

Das folgende Problem ist unter der Bezeichnung "Die Türme von Hanoi" bekanntgeworden. Gegeben sind n Scheiben, alle in der Mitte durchbohrt, alle mit verschiedenem Durchmesser, die so an einem Stab aufgestapelt sind, daß keine Scheibe über einer anderen mit kleinerem Durchmesser liegt, d.h. nach oben nimmt der Durchmesser der Scheiben ab.



Neben dem erwähnten Stab - wir nennen ihn A - stehen noch zwei weitere Stäbe B und C



Die Scheiben **sollen** nun vom Stab A entfernt und an Stab B in derselben Reihenfolge aufeinandergeschichtet werden. Dabei sind die Bewegungsmöglichkeiten wie folgt eingeschränkt:

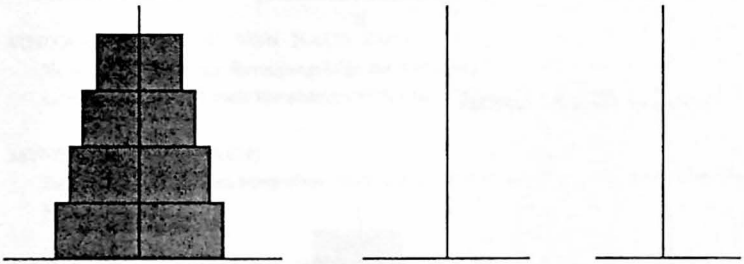
- Es kann immer nur eine Scheibe bewegt werden;
- dabei können Scheiben auch bei Stab C "zwischengeparkt" werden;
- an keiner Stelle darf jeweils eine Scheibe mit größerem Durchmesser auf eine Scheibe mit kleinerem Durchmesser gelegt werden.

Gesucht ist eine Folge von Scheibenbewegungen, welche die gewünschte Umsetzung vom Stab A zu Stab B realisiert. Diese Bewegungsfolge kann nun überraschend einfach mit Hilfe einer rekursiven Funktion ermittelt werden.

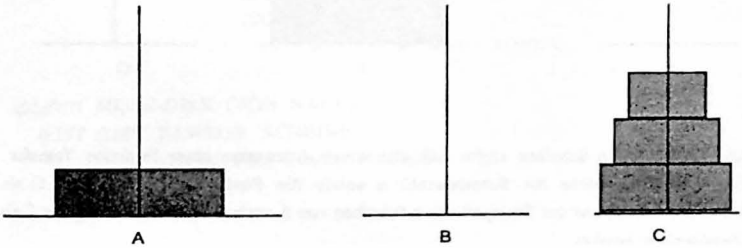
Daß tatsächlich eine rekursive Funktion zur Bestimmung dieser Bewegungsfolge existieren muß, ergibt sich aus folgenden Beobachtungen:

1) Die gesuchte Bewegungsfolge kann für die "einfachen" Fälle mit n = 1 und n = 2 unmittelbar angegeben werden.
2) Die Aufgabe, die Bewegungsfolge für n Scheiben zu bestimmen, kann auf die Aufgabe für n-1 Scheiben zurückgeführt werden und zwar wie folgt:
   Ausgehend von der Anfangssituation, in der alle Scheiben bei A übereinanderliegen, werden die n-1 obersten Scheiben von A nach C umgelagert und dort korrekt abgelegt. Dazu muß das Problem für n-1 Scheiben gelöst werden.

Anfangssituation

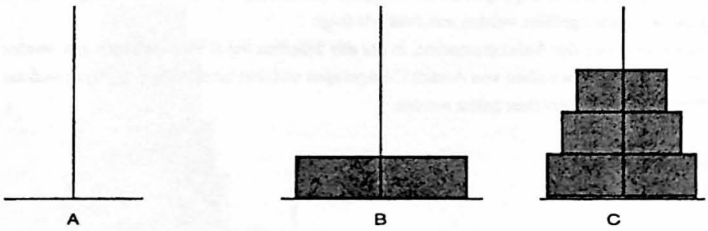


Verlagern der, n-1 oberen Scheiben

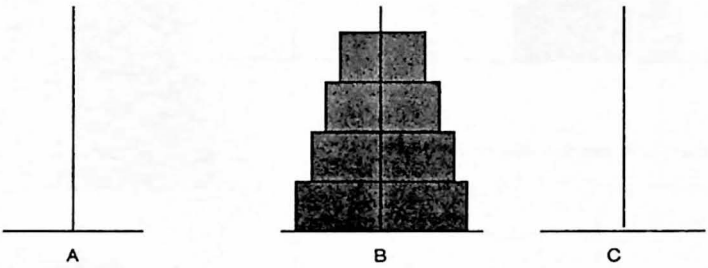


Dann wird die noch bei A verbliebene Scheibe von A nach B gebracht. Anschließend werden die n-1 Scheiben von C nach B umgelagert. Dazu muß nochmals das Problem für n-1 Scheiben gelöst werden.

Verlagern der n-ten Scheibe



Verlagern von n-1 Scheiben



Die Lösung für n Scheiben ergibt sich also durch Auswerten einer Funktion Transfer (n, A, B, C), welche die Scheibenzahl n sowie die Positionen A, B und C als Parameter enthält, und den Transport von n Scheiben von A nach B unter Benutzung von C als Zwischenlager bewirkt.

Transfer (n, A, B, C):
Falls n = 1 Transportiere eine Scheibe von A nach B.
Falls n > 1 Rufe Transfer (n-1, A, C, B)
; rekursiver Aufruf zur Verlagerung der
; n-1 oberen Scheiben von A nach C unter
; Benutzung von B als Zwischenlager
Transportiere eine Scheibe von A nach B
Rufe Transfer (n-1, C, B, A)
; rekursiver Aufruf zur Verlagerung der
; n-1 Scheiben von C nach B unter
; Benutzung von A als Zwischenlager.

Das ist die Lösung ! Die angegebene Funktion Transfer liefert die gesuchte Bewegungsfolge. Die Umsetzung in eine LISP-Funktion gelingt jetzt ohne größere Mühe. Wir definieren dazu zwei Funktionen:

TOWER-OF-HANOI (N VON NACH ZWI)
Sie liefert die gesuchte Bewegungsfolge zur Verlagung von N Scheiben von Position VON nach Position NACH unter Benutzung von Position ZWI als Zwischenlager.

MOVE-DISK (VON NACH)
Sie erzeugt die Beschreibung einer einzelnen Scheibenbewegung von Position VON nach Position NACH.

```
(DEFUN TOWER-OF-HANOI (N VON NACH ZWI)
  (COND ((= N 1) (MOVE-DISK VON NACH))
        (T (APPEND (TOWER-OF-HANOI
                        (- N 1) VON ZWI NACH)
                   (MOVE-DISK VON NACH)
                   (TOWER-OF-HANOI
                        (- N 1) ZWI NACH VON)))))

(DEFUN MOVE-DISK (VON NACH)
  (LIST (LIST 'BEWEGE 'SCHEIBE
              'VON VON 'NACH NACH)))
```

Beispiel:
Die Definitionen für die beiden Funktionen TOWER-OF-HANOI und MOVE-DISK seien abgearbeitet. Wir betrachten dann folgenden **Funktionsaufruf:**

```
-->(TOWER-OF-HANOI 3 'A 'B 'C)
( (BEWEGE SCHEIBE VON A NACH B)    ; Der ermittelte Funktionswert wird
  (BEWEGE SCHEIBE VON A NACH C)    ; nicht unbedingt in dieser übersicht-
  (BEWEGE SCHEIBE VON B NACH C)    ; lichen Gestalt ausgegeben.
  (BEWEGE SCHEIBE VON A NACH B)    ; Möglichkeiten der formatierten Aus-
  (BEWEGE SCHEIBE VON C NACH A)    ; gabe werden in Kapiteln 10 und
  (BEWEGE SCHEIBE VON C NACH B)    ; 13 behandelt.
  (BEWEGE SCHEIBE VON C NACH B) )
```

Der zweimalige Aufruf der Funktion LIST im Rumpf der Funktion
MOVE-DISK ist nicht unbedingt nötig:
MOVE-DISK beschreibt damit Scheibenbewegungen durch Listen der Gestalt
((BEWEGE SCHEIBE VON ...))

Beim Zusammenfassen dieser Listen durch APPEND verschwindet ein Klammerpaar, so daß jede einzelne Bewegung in der Ausgabe geklammert bleibt.
Hätte der Rumpf von MOVE-DISK die Gestalt
(LIST 'BEWEGE 'SCHEIBE 'VON VON 'NACH NACH),

so würden in der Ausgabe für einen Aufruf von TOWER-OF-HANOI die einzelnen Bewegungsbeschreibungen selbst nicht mehr geklammert.

7.4 Zusammenfassung

Rekursivität begegnete uns bisher bei der

- Definition von Datenstrukturen, zum Beispiel:

  Eine Liste ist eine Folge von Elementen, von denen jedes ein Atom oder eine Liste ist

sowie bei der

- Definition von Rechenvorschriften, zum Beispiel:

  Die Umkehrung einer Liste ist im Falle der leeren Liste die leere Liste, ansonsten erhält man sie, indem man die um den Listenkopf verkürzte Liste umkehrt und an diese Umkehrung den alten Listenkopf hinten anfügt.

Diese verbale Beschreibung kann unmittelbar in die formale Definition einer LISP-Funktion überführt werden:

```
(DEFUN OUR-REVERSE (LST)
    (COND ((NULL LST) 'NIL)
          (T (APPEND (OUR-REVERSE (CDR LST))
                     (CONS (CAR LST) 'NIL)))))
```

In beiden Fällen wird eine Eigenschaft unter Bezugnahme auf eben diese Eigenschaft definiert. Wichtig ist dabei nur, daß die Kette der rekursiven Bezüge abbricht.

Frage:
Wo steckt jetzt der Haken bei folgender rekursiven Erklärung:
Eine Katze ist ein Tier, dessen Elterntiere Katzen sind.

KAPITEL 8:

Argumente und Variable

Dieses Kapitel behandelt die Beziehungen von in Funktionen auftretenden Variablen zu außerhalb der Funktion definierten Werten. Im Mittelpunkt steht der Begriff der Umgebung: Darunter verstehen wir die Gesamtheit der zu einem Zeitpunkt aktuellen Zuordnungen zwischen Symbolen und Werten. Mit den sogenannten LET-Ausdrücken wird ein neues Konzept zur Einführung von lokalen Variablenbindungen vorgestellt. Abschließend wird auf sogenannte funktionale Argumente eingegangen. Dabei wird deutlich, daß in LISP auch Funktionen wie Daten behandelt und manipuliert werden können.

8.1 Bindung und Umgebung

Wir kennen bis jetzt zwei Möglichkeiten, eine Zuordnung zwischen Symbolen und Werten herzustellen:

- Die Wertzuweisung mit SETQ und
- die Bindung von Parametern einer Funktion an Argumentwerte.

Sie unterscheiden sich wie folgt:

- Bei der **Wertzuweisung** an ein Symbol wird eine bestehende Zuordnung geändert (wobei wir eine erstmalige Wertzuweisung als Änderung der leeren Zuordnung ansehen).
- Bei der **Bindung** von Parametern wird eine zusätzliche Zuordnung eingerichtet, die nach Abarbeitung des Sprachkonstrukts (hier des Funktionsrumpfes), für das sie eingerichtet wurde, wieder aufgehoben wird.

Die Gesamtheit der zu einem bestimmten Zeitpunkt zugänglichen Zuordnungen zwischen Symbolen und Werten bezeichnet man als (aktuelle) **Umgebung**.
Die Umgebung, die wir bei einem Neustart des LISP-Systems antreffen, heißt **globale Umgebung**, auch **Top-Level-Umgebung**. Sie enthält insbesondere die Zuordnungen zwischen Symbolen und Systemfunktionen.

Als **globalen Wert** eines Atoms bezeichnen wir seinen Wert in der globalen Umgebung; das ist derjenige Wert, den man erhält, wenn man auf das Prompt-Zeichen des Systems den Namen des Atoms eingibt.

Durch Wertzuweisung mit SETQ wird die jeweils aktuelle Umgebung verändert. Dagegen wird durch die zur Auswertung eines Funktionsaufrufs vorgenommene Bindung von Parametern einer Funktion an die Werte von Argumenten nicht etwa eine vorhandene Umgebung geändert, sondern eine neue Umgebung eingerichtet. Die alte Umgebung bleibt dabei erhalten, ist aber nicht mehr aktuell. Sie wird jedoch wieder aktuell, d.h. ihre Zuordnungen werden wieder zugänglich, sobald die Funktionsauswertung, für die die neue Umgebung eingerichtet wurde, abgeschlossen ist. Die neue Umgebung enthält auf jeden Fall die Parameterbindungen. Was ihr sonst noch angehört, hängt davon ab, wie die Werte der freien Variablen festgelegt werden.
Regeln, die festlegen, wann und wo eine Variable zugänglich ist, heißen **Skopierungsregeln** (von "scope" = Gültigkeitsbereich). Sie werden im folgenden behandelt.

Dazu definieren wir

- die **Definitionsumgebung** einer Funktion als diejenige Umgebung, die zum Zeitpunkt ihrer Definition in Kraft ist;
- die **Aufrufumgebung** eines Funktionsaufrufs als diejenige Umgebung, die zum Zeitpunkt des Aufrufs in Kraft ist.

Sodann untersuchen wir einige einfache **Beispiele**:
Wir gehen aus von einer Funktion SUCCESSOR und modifizieren diese anschließend:

```
-->(DEFUN SUCCESSOR (N)                    ; Nachfolger-Funktion
     (+ 1 N)
SUCCESSOR

-->(SUCCESSOR 17)
18
```

Jetzt modifizieren wir SUCCESSOR wie folgt:

```
-->(DEFUN FUNNY-SUCCESSOR (N)
     (SETQ A (+ 1 N))
     (AUX-SUCCESSOR)
FUNNY-SUCCESSOR

-->(DEFUN AUX-SUCCESSOR ( )
     (+ 1 N))
AUX-SUCCESSOR
```

```
-->(SETQ N 10)
10

-->(FUNNY-SUCCESSOR 17)
11

-->A
18
```

Wir betrachten die Auswertung des Aufrufs (FUNNY-SUCCESSOR 17):
Zuerst wird für die lokale Variable N von FUNNY-SUCCESSOR eine Bindung an den Wert 17 eingerichtet. Diese ist offensichtlich zugänglich, solange die Auswertung im textuellen Bereich des Rumpfs von FUNNY-SUCCESSOR abläuft, also während der Abarbeitung der SETQ-Form, aber nicht mehr während der Abarbeitung des AUX-SUCCESSOR-Aufrufs. Bezüglich AUX-SUCCESSOR ist N eine freie Variable. Während der Auswertung von (AUX-SUCCESSOR) wird offensichtlich für N der Wert aus der Definitionsumgebung von AUX-SUCCESSOR zugrundegelegt. Dieser Wert ist aber gleich 10. Nach Addition von 1 übergibt AUX-SUCCESSOR den Wert 11 an die aufrufende Funktion FUNNY-SUCCESSOR, die diesen ihrerseits als Wert ihrer letzten Form übergibt.

Bei älteren Dialekten von LISP, z.B. MACLISP, wäre die Auswertung anders gelaufen. Dort wäre während der Auswertung von AUX-SUCCESSOR für N der Wert aus der Aufrufumgebung, also 17, zugrundegelegt worden; somit hätte AUX-SUCCESSOR den Wert 18 übergeben, der dann auch Wert von (FUNNY-SUCCESSOR 17) geworden wäre.

Wir halten fest:

Einige ältere Dialekte von LISP orientieren sich bei der Zuordnung von freien Variablen einer Funktion an der Aufrufumgebung, bilden also die neue Umgebung aus den Parameterbindungen und den Zuordnungen der freien Variablen gemäß der Aufrufumgebung. Man spricht dabei von **dynamischer Skopierung**, d.h. dynamischer Bestimmung des Gültigkeitsbereichs (= scope).

Heute hat sich jedoch die Überzeugung durchgesetzt, daß es natürlicher ist, die Bindungen für freie Variable aus der Definitionsumgebung der Funktion zu entnehmen. Man spricht dabei von **lexikalischer Skopierung**. Sie ist auch in COMMON LISP der "Normalfall". Nur für besonders deklarierte "spezielle" Variable und spezielle Sprachkonstrukte gilt dynamische Skopierung. In unserem Beispiel erreicht man dies mit einer zusätzlichen Variablendeklaration wie folgt:

```
-->(DEFUN ANOTHER-SUCCESSOR (N)
      (DECLARE (SPECIAL N))
      (AUX-ANOTHER-SUCCESSOR))
ANOTHER-SUCCESSOR

-->(DEFUN AUX-ANOTHER-SUCCESSOR ( )
      (DECLARE (SPECIAL N))
      (+ 1 N)
AUX-ANOTHER-SUCCESSOR

-->(ANOTHER-SUCCESSOR 17)
18
```

Die zusätzliche Variablendeklaration erfolgt zwischen Parameterliste und Rumpf. Sie sichert dynamische Skopierung für die so deklarierten Variablen. Wir gehen darauf hier nicht ein, weil es den Rahmen dieser Einführung sprengt. Grundsätzlich sollten freie Variable in Funktionen nur mit größter Vorsicht eingesetzt werden.
Spezielle Variable der oben gezeigten Art werden auch nur selten eingesetzt. Ein Spezialfall dieser "speziellen" Variablen, die **globalen** Variablen wird dagegen häufig benutzt. Sie werden im nächsten Abschnitt behandelt.

8.2 Globale Variable

Die Verwendung von freien Variablen ist dann sinnvoll, wenn Information über den Zustand eines Programms von vielen Funktionen benutzt und gegebenenfalls auch verändert werden soll.
Manchmal soll eine Variable für soviele Funktionen zugänglich sein, daß man sie gerne für alle als "speziell" deklarieren möchte, ohne diese Deklaration in jeder Funktion erneut vornehmen zu müssen. Dieser Aufgabe dienen die **globalen Variablen** , das sind die in der globalen Umgebung, also auf dem Top-Level, definierten Variablen.
Verschiedene globale Variable werden schon vom System zur Verfügung gestellt.
Vom Benutzer können globale Variable auf verschiedene Arten eingeführt werden.

1) Implizit durch Wertzuweisung mit SETQ auf dem Top-Level
   oder
2) explizit durch Aufruf der speziellen Form DEFVAR mit dem
   Aufrufmuster:

(DEFVAR <Symbol> <Anfangswert>)

Das erste Argument ist ein Symbol, das als Variable eingeführt wird.
Das zweite Argument ist ein Ausdruck; es liefert einen Anfangswert für die Variable, falls diese noch keinen Wert hat. Dieses zweite Argument muß nicht unbedingt vorhanden sein; wir sagen dazu, es ist optional.

Auswertung von DEFVAR:
Als **Seiteneffekt** wird das im ersten Argument gegebene Symbol als globale Variable eingeführt; dies gilt auch dann, wenn DEFVAR nicht auf dem Top-Level aufgerufen wird.

Das zweite Argument wird nur dann ausgewertet, wenn die globale Variable noch keinen Wert hat. In diesem Fall wird ihr der ermittelte Wert als Anfangswert zugewiesen. Hatte sie schon einen Wert, so behält sie diesen.

Als **Wert** der DEFVAR-Form wird in jedem Fall das Symbol übergeben.

Um globale Variable besser als solche erkennen zu können, werden sie häufig mit Symbolen bezeichnet, die mit dem Stern beginnen und schließen. Diese Symbole sehen demnach aus wie

COUNT oder *GLOBAL*

Der Wert einer mit DEFVAR deklarierten Variable kann durch eine Wertzuweisung mit SETQ, aber nicht durch eine erneute Deklaration mit DEFVAR verändert werden.

Beispiel:

```
-->(DEFVAR *GLOBAL* (SETQ OKTOBERFEST 'BEER))
*GLOBAL*

-->*GLOBAL*
BEER

-->OKTOBERFEST                    ;  zeigt, daß zweites Argument von
BEER                              ;  DEFVAR ausgewertet wurde
```

```
-->(DEFVAR *GLOBAL* (SETQ WEINFEST 'WEIN))
*GLOBAL*

-->*GLOBAL*                          ; zweite Deklaration war wirkungslos,
BEER                                 ; weil *GLOBAL* schon Wert hatte

-->WEINFEST                          ; zeigt, daß zweites Argument im zweiten
ERROR: WEINFEST has no global value  ; DEFVAR-Aufruf nicht mehr ausge-
                                     ; wertet wurde

-->(SETQ *GLOBAL* 'WODKA)            ; Wertzuweisung verändert Wert
WODKA

-->*GLOBAL*
WODKA
```

Ein Beispiel

mit einer globalen Zählvariablen *COUNT* zeigt, daß die Verwendung von freien globalen Variablen recht sinnvoll sein kann. Wir definieren hier eine neue Version von CONS, bei der in einer globalen Zählvariablen die Anzahl der Anwendungen von CONS gezählt wird, und verwenden diese dann auch für eine Neudefinition von APPEND. Da CONS eine aufwendige Operation ist, kann eine derartige Überwachung Bedeutung haben.

```
--->(DEFVAR *COUNT*)                 ; Definition der globalen Variablen
*COUNT*

-->(DEFUN COUNT-CONS (X Y)           ; Wert von *COUNT* wird bei jeder
     (SETQ *COUNT* (+ 1 *COUNT*))    ; Anwendung von CONS durch Seiten-
     (CONS X Y))                     ; effekt um 1 erhöht
COUNT-CONS

-->(DEFUN COUNT-APPEND (X Y)         ; Wirkt wie APPEND, zählt aber zu-
     (COND ((NULL X) Y)              ; sätzlich die Anwendungen von CONS
           (T (COUNT-CONS (CAR X)
                          (COUNT-APPEND (CDR X) Y)))))
COUNT-APPEND
```

```
-->(SETQ *COUNT*  0)                    ; Wertzuweisung
0

-->(COUNT-APPEND '(A B C) '(D E F))
(A B C D E F)

-->*COUNT*
3                                       ; Es wurde 3mal "geCONSt"

-->(SETQ *COUNT* 0)                     ; Rücksetzung von *COUNT* auf 0.
0
```

8.3 Einrichtung neuer Variablenbindungen mit LET-Ausdrücken

Lokale Variable, also Variable die nur in einem beschränkten Teil eines Programms Bedeutung haben, können auf verschiedene Weisen eingeführt werden.
Wir kennen bis jetzt eine, nämlich durch Aufnahme der Variablen in die Parameterliste einer Funktion.
Eine andere besteht darin, Variable in einem sogenannten LET-Ausdruck einzuführen. Ein solcher hat das **Aufrufmuster:**

```
(LET (  (<Par-1> <Ausdruck-1>)
        .
        .
        .
        (<Par-n> <Ausdruck-n>) )
    <Rumpf>)
```

Auf das Wortsymbol LET folgt eine Liste von Paaren, welche die lokalen Bindungen festlegen, unter denen der nachfolgende Rumpf auszuwerten ist. Dabei werden die Symbole Par-1 bis Par-n als lokale Variable eingeführt; sie werden, wie anschließend beschrieben, an die Werte von Ausdruck-1 bis Ausdruck-n gebunden. Der Rumpf besteht aus einer Folge von null, einer oder mehreren auswertbaren Formen.

Auswertung:
Zunächst werden die Ausdrücke Ausdruck-1 bis Ausdruck-n der Reihe nach ausgewertet. Wenn sie alle ausgewertet sind, werden die Variablen Par-1 bis Par-n an die für Ausdruck-1 bis Ausdruck-n ermittelten Werte gebunden. Bei Auswertung etwa von Ausdruck-3 kann also der Wert von Par-1 oder Par-2 nicht benutzt werden. Man sagt deshalb auch, die Variablen werden "**parallel**" an die Werte der zugehörigen Ausdrücke gebunden.
Unter diesen Bindungen wird dann der Rumpf abgearbeitet. Dazu werden die Formen, aus denen der Rumpf besteht, der Reihe nach ausgewertet. Der Wert der letzten Form wird Wert der LET-Form. Besteht der Rumpf aus null Formen, so wird NIL der Wert der LET-Form.
Nach Auswertung des Rumpfes wird die Bindung der Variablen Par-1 bis Par-n aufgehoben.

Seiteneffekte:
Nur die durch Auswertung von Ausdruck-1 bis Ausdruck-n und Auswertung des Rumpfes bewirkten Seiteneffekte.

Die für den **LET-Ausdruck** beschriebene Auswertung entspricht offensichtlich der Auswertung der folgenden **Lambda-Anwendung.**

```
((LAMBDA  (<Par-1> . . . <Par-n>)
          <Rumpf>)
          <Ausdruck-1> . . . <Ausdruck-n>)
```

Beispiel:

```
-->(SETQ X 2)
2

-->(LET    ((X  (+ 1 X))
           (Y  (+ 2 X)))
          (LIST X Y))
(3 4)
```

Für die Auswertung des Rumpfes, er enthält nur die Form (LIST X Y) , wurde eine neue Umgebung mit den lokalen Variablen X und Y eingerichtet. Die Auswertung der Ausdrücke (+ 1 X) und (+ 2 X) für die Anfangswerte der lokalen Variablen wird noch vollständig in der globalen Umgebung durchgeführt. Wir machen insbesondere darauf aufmerksam, daß zum Zeitpunkt der Auswertung des Ausdrucks für Y die Variable X noch ihre globale Bedeutung hat. Andernfalls hätte das zweite Listenelement in der Ausgabe nicht den Wert 4.

Der LET-Ausdruck im obigen Programmstück kann ersetzt werden durch die Lambda-Anwendung

```
-->((LAMBDA (X Y) (LIST X Y)) (+ 1 X) (+ 2 X))
(3 4)
```

Tatsächlich wird im LISP-System die Auswertung von LET-Ausdrücken immer auf die Auswertung einer äquivalenten Lambda-Anwendung zurückgeführt.

LET-Ausdrücke sind zweifellos übersichtlicher als entsprechende Lambda-Anwendungen; die Zuordnung zwischen Parametern und Argumenten ist leichter erkennbar.
Die Bedeutung von Lambda-Ausdrücken liegt insbesondere in ihrer Verwendbarkeit als funktionale Argumente, die wir im übernächsten Abschnitt kennenlernen werden.

8.4 Beispiele zur Skopierung

Fragen der Skopierung, also der Zugänglichkeit von Variablen gehören eigentlich bei jeder Programmiersprache zu den schwierigsten Problemen.

Deshalb sollen die wichtigsten Skopierungsregeln von LISP an einigen weiteren Beispielen nochmals verdeutlicht werden.
Wir gehen aus von einem **Beispiel**, in dem durch Einführung einer lokalen Variablen eine nichtlokale Variable gleichen Namens vorübergehend unzugänglich wird:

```
->(LET    ((X 2)
           (Z 5))
          (LET ((X (+ 1 X))
                (Y (+ 2 X)))
               (LIST X Y Z)))
(3 4 5)
```

Zur Abarbeitung des Rumpfs der äußeren LET-Form wird eine Umgebung mit lokalen Variablen X und Z eingerichtet, die dort an 2 bzw. 5 gebunden werden.

Im Zuge der Auswertung dieses Rumpfs wird dann die innere LET-Form ausgewertet. Dabei wird eine neue Umgebung eingerichtet, deren lokale Variable X und Y heißen. Ihre Anfangswerte werden aber noch in der bisherigen Umgebung ermittelt, in der X an 2 gebunden war.

Nach Einrichtung der neuen Umgebung sind deren lokale Variable X und Y an 3 bzw. 4 gebunden. Aufgrund der Bindung von X in der neuen Umgebung ist die Bindung der gleichnamigen Variablen X der alten Umgebung nicht mehr zugänglich. An der Bindung der Variablen Z hat sich dagegen nichts geändert. Sie bleibt auch in der neuen Umgebung zugänglich, da keine lokale Variable gleichen Namens eingeführt worden ist.

Analoge Überlegungen gelten, wenn wir die LET-Ausdrücke im obigen Beispiel durch entsprechende Lambda-Anwendungen ersetzen. Sie gelten auch, wenn wir diese durch Definitionen und Aufrufe benannter Funktionen ersetzen; doch treten dabei zusätzliche Gesichtspunkte auf:

```
-->(DEFUN FU1 (X Z)
     (DEFUN FU2 (X Y)
        (LIST X Y Z))
     (FU2 (+ 1 X) (+ 2 X)))
FU1

-->(FU1 2 5)
(3 4 5)
```

Die DEFUN-Form zur Definition von FU2 ist Teil des Rumpfs von FU1; sie wird nicht bei Auswertung der DEFUN-Form für FU1, sondern erst bei Aufruf von FU1 abgearbeitet. Da die Auswertung einer DEFUN-Form grundsätzlich eine globale Zuordnung des Funktionsnamens an die Funktion bewirkt, kann FU2 nach einer Auswertung von FU1 auch außerhalb des Rumpfs von FU1 aufgerufen werden. FU2 enthält aber noch die freie Variable Z. Deshalb betrachten wir nach Abarbeitung des obigen Aufrufs von FU1 die Abarbeitung des folgenden Programmstücks:

```
-->(SETQ Z 11)
11

-->(FU2 6 7)
(6 7 5)
```

Zur Auswertung von FU2 wurde für die freie Variable Z natürlich nicht der Wert 11 aus der globalen Umgebung herangezogen, sondern die Bindung an 5 aus der Definitionsumgebung zugrunde gelegt. So schreibt es die lexikalische Skopierung vor.

Im Falle dynamischer Skopierung wäre für Z der Wert 11 aus der Aufrufumgebung zugrundegelegt worden.

In einem weiteren Beispiel betrachten wir nun das Auftreten von DEFVAR-Deklarationen im Rumpf einer Funktion.

```
-->(DEFUN FU (Z)
     (DEFVAR Z 12)
     (DEFVAR Y Z))
FU
```

Beim Auswerten eines Aufrufs der Funktion FU werden die globalen Variablen Z und Y deklariert:

```
-->(FU 3)
Y
```

Als Wert wird der Wert der letzten Form aus dem Rumpf übergeben.
Jetzt fragen wir nach den Werten der globalen Variablen Z und Y:

```
-->Z
12

-->Y
3
```

Man erkennt:
Während der Abarbeitung des Rumpfes von FU wird eine globale Variable Z mit dem Wert 12 deklariert. Dies gelingt, obwohl der Rumpf in einer Umgebung abgearbeitet wird, die eine lokale Variable Z mit einer Bindung an 3 enthält. Die globale Variable Z ist aber während der weiteren Abarbeitung des Rumpfs nicht mehr zugänglich. Sie wird dort durch die lokale Variable Z und deren Bindung an 3 überdeckt. Deshalb wird bei Auswertung von (DEFVAR Y Z) für Z die lokale Bindung an 3 zugrundegelegt. So kommt die globale Variable Y zum Wert 3.

8.5 Funktionale Argumente

Eine der wesentlichen Eigenschaften von LISP ist, daß LISP-Funktionen nicht nur dieselbe Gestalt wie LISP-Daten haben, sondern auch wie Daten behandelt und verarbeitet werden können. Insbesondere ist es möglich, Ausdrücke zu bilden, die Funktionsbeschreibungen als Wert haben.

Solche Ausdrücke sind zum Beispiel die COND-Form

```
(COND ((< X Y) 'ADJOIN)
      (T       'MEMBER))
```

sowie das Symbol FN nach Auswertung der Wertzuweisung

```
(SETQ FN (COND ((< X Y) 'ADJOIN)
               (T       'MEMBER)))
```

Jetzt stellt sich natürlich die Frage, wie die durch den obigen COND-Ausdruck oder durch das Symbol FN letztlich dargestellte Funktion ausgewertet werden kann.

Dazu stellen wir zuerst fest:
Ein Ausdruck, der erst noch auszuwerten ist, kann in einem Funktionsaufruf der Gestalt

(<Funktionsbeschreibung> <Arg-1> ... <Arg-n>)

nicht an Stelle der Funktionsbeschreibung verwendet werden. Eine Auswertung ist grundsätzlich nur in Argumentposition möglich. Dazu werden in erster Linie die Systemfunktionen FUNCALL und APPLY verwendet.

FUNCALL hat das **Aufrufmuster:**

(FUNCALL <Funktionales Objekt> <Arg-1> ... <Arg-n>)

Das erste Argument ist ein sogenanntes funktionales Objekt; darunter verstehen wir einen Ausdruck, dessen Wert eine Funktionsbeschreibung ist. Zur Auswertung der dadurch gegebenen Funktion werden die weiteren Argumente herangezogen.

Auswertung:

Zunächst werden die Argumente von FUNCALL der Reihe nach ausgewertet. Der Wert des ersten Arguments ergibt die Funktion, die eigentlich auszuwerten ist. Die Argumente für ihre Auswertung sind gegeben durch die weiteren Argumente Arg-1 bis Arg-n von FUNCALL. Zur Auswertung werden - wie bei normalen Funktionsaufrufen - die Parameter der Funktion an die Werte dieser Argumente gebunden.

Wert

der FUNCALL-Form ist das Ergebnis der Anwendung der durch das erste Argument gegebenen Funktion auf die Werte der restlichen Argumente.

Seiteneffekt:

Keiner, mit Ausnahme der Seiteneffekte, die durch die Auswertung der n+1 Argumente und die Auswertung der Funktion verursacht werden.

Beispiele:

```
-->(SETQ L '(A B C) X 3 Y 5)
5
-->(LET ((FN (COND ((< X Y) 'ADJOIN)
                   (T       'MEMBER))))
     (FUNCALL FN 'D L))
(D A B C)
-->(FUNCALL 'ADJOIN 'D L)
(D A B C)
```

Zur Auswertung einer durch ein funktionales Objekt gegebenen Funktion, kann auch die Funktion APPLY verwendet werden. Sie übernimmt ähnlich wie FUNCALL ein funktionales Objekt als Argument, kann aber die Argumente für das funktionale Objekt sowohl einzeln als auch in Listen übernehmen. Die Funktion APPLY ist also flexibler einsetzbar als FUNCALL. Sie hat das

Aufrufmuster:

(APPLY <Funktionales Objekt> <Arg-1> ... <Arg-n> <Arg-(n+1)>)

Von den n+2 Argumenten von APPLY ist das erste ein funktionales Objekt; dann folgen $n \geq 0$ Argumente Arg-1 bis Arg-n und ein weiteres Argument Arg-(n+1), dessen Wert eine Liste sein muß. Die Werte von Arg-1 bis Arg-n und die Elemente der Liste bilden in der gegebenen Reihenfolge die Argumentwerte zur Auswertung der durch das funktionale Objekt gegebenen

Funktion. Im Falle n = 0 folgt auf das funktionale Objekt nur noch ein Argument, dessen Wert eine Liste sein muß; ihre Elemente liefern in diesem Fall alle Argumentwerte für die Auswertung der durch das funktionale Objekt gegebenen Funktion.

Auswertung von APPLY:
Zuerst werden alle n+2 Argumente ausgewertet.
Der Wert des funktionalen Objekts liefert die Funktion, die eigentlich auszuwerten ist. Zur Auswertung dieser Funktion werden deren Parameter gebunden an die Werte der weiteren Argumente Arg-1 bis Arg-n von APPLY mit Ausnahme des letzten Arguments Arg-(n+1) und an die Elemente der Liste, die Wert von Arg-(n+1) ist.

Wert
der APPLY-Form ist das Ergebnis der beschriebenen Funktionsauswertung.

Seiteneffekt:
Keiner, mit Ausnahme der Seiteneffekte, die durch die Auswertung der n+2 Argumente und die Auswertung der Funktion verursacht werden.

Beispiele:

```
-->(APPLY '+ '(1 2 3 4 5 6))
21
-->(APPLY '+ 1 2 3 '(4 5 6))
21
-->(APPLY '+ '1 '2 '3 '4 '5 '6 ())
21
```

Sehr häufig hat APPLY nur zwei Argumente, nämlich das funktionale Objekt und ein Argument, dessen Wert eine Liste ist. Das entspricht oben dem Fall n = 0. Die Argumente für die durch das funktionale Objekt gegebene Funktion sind dann allein durch die Elemente der Liste gegeben.

Der Unterschied zwischen einer Funktionsauswertung mit FUNCALL bzw. APPLY und einem normalen Funktionsaufruf besteht darin, daß die auszuwertende Funktion bei FUNCALL bzw. APPLY erst durch Auswertung eines Arguments ermittelt werden muß, während sie bei einem "normalen" Funktionsaufruf unmittelbar gegeben ist.
FUNCALL kann eingesetzt werden, wenn die Argumente für die auszuwertende Funktion einzeln vorliegen; APPLY ist anzuwenden, wenn die Argumente alle oder zum Teil als Elemente einer Liste anfallen.

Weitere Beispiele:

```
-->(APPLY 'ADJOIN 'X '(A B C) '( ))
(X A B C)

-->(APPLY 'ADJOIN 'X '(A B C)))
(X A B C)

-->(APPLY 'ADJOIN 'X '((A B C)))
(X A B C)
```

8.6 Funktionale Objekte

In der Mathematik bezeichnet man eine Funktion, deren Wertebereich wiederum aus Funktionen besteht, üblicherweise als Funktional. Die Auswertung eines Funktionals liefert also eine Funktion. Dies ist der Grund, weshalb wir einen LISP-Ausdruck, dessen Auswertung eine Funktionsbeschreibung liefert, als funktionales Objekt bezeichnen.

Wir haben im vorhergehenden Abschnitt gesehen wie Funktionen, die durch funktionale Objekte gegeben sind, mit Hilfe von FUNCALL oder APPLY ausgewertet werden können. Die bisher aufgetretenen funktionalen Objekte waren von besonders einfacher Gestalt. Funktionale Objekte haben häufig auch die Gestalt eines Aufrufs der Systemfunktion SYMBOL-FUNCTION oder FUNCTION.

Die Systemfunktion SYMBOL-FUNCTION dient dazu, die einem Symbol zugeordnete globale Funktionsdefinition zu ermitteln und zwar mit einem Funktionsaufruf der Gestalt

(SYMBOL-FUNCTION <Symbol>)

Jeder Ausdruck der Gestalt

(SYMBOL-FUNCTION <Symbol>)

ist ein funktionales Objekt. Seine Auswertung liefert die dem Symbol global zugeordnete Funktion.

Beispiele für funktionale Objekte sind demnach

(SYMBOL-FUNCTION 'MEMBER)

und

(SYMBOL-FUNCTION FN)

wenn dem Symbol FN etwa durch

```
(SETQ FN (COND ((< X Y) 'ADJOIN)
               (T       'MEMBER)))
```

als Wert eine Funktionsbeschreibung zugewiesen ist.

Eine andere Möglichkeit, ein funktionales Objekt zu bilden, ist die Anwendung einer Systemfunktion FUNCTION auf einen Lambda-Ausdruck; sie führt auf einen Ausdruck der Gestalt

(FUNCTION <Lambda-Ausdruck>)

Wert
dieser Form ist die sogenannte **Closure** des Lambda-Ausdrucks. Das ist die durch den Lambda-Ausdruck dargestellte Funktion zusammen mit der korrekten Bindung der freien Variablen, entsprechend den Regeln der lexikalischen Skopierung.

Da ein Lambda-Ausdruck an sich schon die Beschreibung einer Funktion darstellt, fragen wir uns, worin hier die Wirkung der Funktion FUNCTION und die Bedeutung der resultierenden Closure besteht. Um hier eine Antwort geben zu können, betrachten wir das folgende Programmstück

```
-->(DEFUN ADDER (X) (FUNCTION (LAMBDA (Y) (+ X Y))))
ADDER
```

Der Wert eines Aufrufs von ADDER , etwa von (ADDER 3) , ist eine derartige Closure; sie stellt eine Funktion dar, die 3 zu ihrem Argument addiert. Die externe Darstellung des Wertes von (ADDER 3) bringt dies im allgemeinen nur unvollständig zum Ausdruck:

```
-->(ADDER 3)
#<Interpreted Function:  (LAMBDA (Y) (+ X Y))>
```

Jetzt ersetzen wir versuchsweise die Anwendung von FUNCTION auf den Lambda-Ausdruck durch eine Anwendung von QUOTE

```
-->(DEFUN ADDERQ (X)
            (QUOTE (LAMBDA (Y) (+ X Y))))
ADDERQ
```

und betrachten dann die Wertzuweisungen

```
-->(SETQ ADD3 (ADDER 3))
#<Interpreted Function: (LAMBDA (Y) (+ X Y))>
```

und

```
-->(SETQ ADDQ3 (ADDERQ 3))
(LAMBDA (Y) (+ X Y))
```

Sie führen zu unterschiedlichen Ausgaben, die darauf hindeuten, daß ADD3 und ADDQ3 unterschiedliche Funktionen als Wert haben. Dies wird auch bestätigt durch die folgenden Aufrufe:

```
-->(FUNCALL ADD3 6)
9

-->(FUNCALL ADDQ3 6)
ERROR: X has no global value
```

Bei Auswertung der ADD3 zugeordneten Funktion wird die Bindung ihrer freien Variablen X an 3 berücksichtigt; diese Bindung besteht aber offensichtlich nicht bei Auswertung der ADDQ zugeordneten Funktion.
Zur Klärung versuchen wir die beiden Aufrufe noch einmal, aber jetzt unter Voranstellen einer Wertzuweisung an X in der globalen Umgebung:

```
-->(SETQ X 5)
5

-->(FUNCALL ADD3 6)
9
```

```
-->(FUNCALL ADDQ3 6)
11
```

Offensichtlich geschieht hier folgendes:
Bei Auswertung der ADDQ3 zugeordneten Funktion wird die Bindung für X der globalen Umgebung entnommen. Bei Auswertung der ADD3 zugeordneten Funktion wird der Wert für X nicht der globalen Umgebung entnommen, sondern derjenigen Umgebung, in der der Lambda-Ausdruck definiert wurde.

Dies illustriert die **Bedeutung der Closure**, die man durch die Anwendung von FUNCTION auf den Lambda-Ausdruck erhält: Die Closure der durch den Lambda-Ausdruck gegebenen Funktion ist anschaulich die durch den Lambda-Ausdruck gegebene Funktion einschließlich der korrekten Bindung der freien Variablen entsprechend ihrer Bedeutung in der Definfitionsumgebung des Lambda-Ausdrucks.

Wir wollen das Problem der Wirkung von FUNCTION auch noch von anderer Seite beleuchten. Dazu modifizieren wir die Funktion ADDER noch einmal:

Wir ersetzen den Aufruf

```
(FUNCTION (LAMBDA (Y) (+ X Y)))
```

durch

```
(DEFUN AHA (Y) (+ X Y))
```

und betrachten folgende Aufrufe:

```
-->(DEFUN ADDERF (X)
        (DEFUN AHA (Y) (+ X Y)))
ADDERF

-->(SETQ ADDF ADDERF (3))                ; dies impliziert die Auswertung der
AHA                                      ; DEFUN-Form für AHA
```

Jetzt ist ADDF ein funktionales Objekt, dessen Wert AHA eine Funktionsbeschreibung ist. Bei Auswertung von AHA wird aufgrund der lexikalischen Skopierung für Z die Bindung aus der Definitionsumgebung zugrundegelegt.

Dies zeigt, daß die bei Anwendung von FUNCTION auf einen LAMBDA-Ausdruck resultierende Closure die lexikalische Skopierung gerade in der Form realisiert, wie sie bei einer mit DEFUN definierten Funktion gegeben ist.

Da funktionale Objekte der oben betrachteten Gestalt häufig auftreten, verwendet man - ähnlich wie bei QUOTE - Abkürzungen:

(SYMBOL-FUNCTION '<Symbol>) wird abgekürzt zu #'<Symbol>

(FUNCTION <Ausdruck>) wird abgekürzt zu #'<Ausdruck>

Ist das Argument von FUNCTION ein Symbol, so ist

(FUNCTION <Symbol>) äquivalent zu (SYMBOL-FUNCTION '<Symbol>)

Das folgende Programmstück illustriert die Bedeutung der Funktion FUNCTION nochmals an einer Variante des obigen Beispiels:

```
-->(SETQ X 5)
5

-->(DEFUN FU1(X)
          (FUNCALL '(LAMBDA (Y) (+ X Y)) 6))
FU1

-->(DEFUN FU2(X)
          (FUNCALL #'(LAMBDA (Y) (+ X Y)) 6))
FU2

-->(FU1 3)
11
```

```
-->(FU2 3)
9
```

Die Variable X ist bezüglich der Funktionen FU1 und FU2 jeweils gebunden, bezüglich des im Rumpf dieser Funktionen auftretenden Lambda-Ausdrucks aber jeweils frei. Beim Aufruf von FU1 bzw. FU2 wird jeweils eine neue Umgebung eingerichtet, die eine Bindung von X an den Wert 3 beinhaltet. Bei Auswertung des Rumpfes von FU1 bzw. FU2 muß jeweils der darin enthaltene Lambda-Ausdruck ausgewertet werden. Dabei wird der Wert für X im Falle von FU1 aus der globalen Umgebung (= Top-Level-Umgebung) geholt, im Falle von FU2 aus der Umgebung, in der der Lambda-Ausdruck definiert ist. Dies wird erreicht durch die oben besprochene Einrichtung der Closure aufgrund der Anwendung der Funktion FUNCTION.

8.7 Zusammenfassung

LISP ist eine funktionale Sprache; im Mittelpunkt steht demnach die Anwendung von Funktionen. Diese Lektion beschäftigt sich mit Argumenten und Variablen bei Funktionen. Die Definition von Funktionen ist eine der wichtigsten Möglichkeiten, Variablen mit beschränktem Gültigkeitsbereich einzuführen. Den dabei auftretenden Problemen war die erste Hälfte dieser Lektion gewidmet.

Charakteristisch für LISP ist, daß LISP-Funktionen und LISP-Daten dieselbe Gestalt haben. Deshalb können Ausdrücke, die Funktionen darstellen, auch wie Daten manipuliert und verarbeitet werden. Sie können selbst wieder Argumente von Funktionen oder Ergebnisse von Funktionsanwendungen sein.

KAPITEL 9:

Kontrollstrukturen

LISP stellt wie jede Programmiersprache Strukturen für die Organisation von Programmen zur Verfügung. Einige regeln den Zugriff auf Variable; wir sprechen dabei von Umgebungsstrukturen. Andere regeln die Ablaufstruktur, wir sagen auch den Kontrollfluß von Programmen; sie heißen Kontrollstrukturen. Ihnen ist dieses Kapitel gewidmet.

Das zentrale Konzept bei der Konstruktion von LISP-Programmen ist die Anwendung von Funktionen (function application): Operationen werden beschrieben als Anwendungen von Funktionen auf ihre Argumente. Gewöhnlich ist ein LISP-Programm einfach eine umfangreiche Kollektion von Funktionen, von denen jede eine Operation implementiert.
Diese Funktionen rufen sich gegenseitig auf und bauen damit umfangreichere Operationen aus einfacheren auf. LISP-Funktionen können sich dabei selbst rekursiv aufrufen, direkt oder indirekt. Insbesondere können Argumente und Werte von Funktionen selbst wieder Funktionen - also Programme - sein. Dies ist auch der Grund, warum man LISP als funktionale Sprache bezeichnet. Funktional heißt nämlich eine Funktion, deren Argumente und Werte selbst wieder Funktionen sind.

Die Auswertung und Abarbeitung von Programmen ist also in erster Linie durch die Aufrufbeziehungen zwischen den das Programm bildenden Funktionen bestimmt, weniger durch Anweisungen. Trotzdem enthält auch LISP viele Operationen, die Seiteneffekte bewirken und benötigt deshalb auch Sprachkonstrukte zur Steuerung der Reihenfolge, in der Seiteneffekte produziert werden.
Die Systemfunktion PROGN zum Beispiel erfüllt ähnliche Aufgaben wie BEGIN-END-Blöcke in imperativen Sprachen (von "imperare" = befehlen, anweisen) wie ALGOL oder PASCAL.
Sie bewirkt die sequentielle Auswertung einer Folge von Formen, wobei am Ende nur der Wert der letzten Form von Bedeutung ist. Viele LISP-Kontrollstrukturen praktizieren eine sequentielle Auswertung von Programmstücken, man sagt deshalb auch, sie enthalten ein "implizites PROGN".

Zur Programmierung von Schleifen verfügt LISP heute über ein sehr allgemeines und mächtiges Konzept in Gestalt der DO-Form sowie über eine Reihe von Sprachkonstrukten, die spezielle Formen der wiederholten Anwendung von Funktionen unterstützen.

9.1 PROG-Formen: PROG, PROG1 und PROGN

LISP enthält in der sogenannten PROG-Form ein Sprachelement, das einen Programmierstil ermöglicht wie er in imperativen Sprachen (Beispiele sind alle Sprachen der ALGOL-PASCAL-FORTRAN-Familie) üblich ist.
Kennzeichnend für diesen Programmierstil ist die Auffassung von Programmen als Folgen von Anweisungen, die entsprechend der gegebenen Ablaufstruktur abgearbeitet werden. Diese "imperative" Programmierung steht im Gegensatz zum Konzept der funktionalen Programmierung, die das herausragende Kennzeichen der Sprache LISP darstellt. Trotzdem wurde die PROG-Form früher häufig verwendet, nicht nur weil sie es ermöglichte, einen vertrauten Programmierstil beizubehalten, sondern vor allem weil sie lange Zeit die einzige Möglichkeit bot, Schleifen explizit zu formulieren. Für Schleifen steht heute in LISP mit der DO-Form ein exzellentes Werkzeug zur Verfügung. Die PROG-Form ist damit überflüssig geworden und wird hier nicht behandelt. Da sie auch dem funktionalen Charakter der Sprache LISP nicht entspricht, wird sie heute kaum noch verwendet.
Natürlich müssen in vielen Fällen Folgen von Formen sequentiell abgearbeitet werden. Wir erinnern uns hier an verschiedene Sprachkonstrukte, die ein Programmstück enthielten, das wir als Rumpf bezeichnet haben, insbesondere die Definition von Funktionen mit DEFUN oder LAMBDA
(DEFUN <Funktionsname> <Parameterliste> <Rumpf>)
bzw.
(LAMBDA <Parameterliste> <Rumpf>)

Der Rumpf ist in diesen (und anderen) Fällen eine Folge von null oder mehr Formen, die sequentiell auszuwerten sind, wobei der Wert der letzten Form jeweils an die umfassende Form übergeben wird.

Gelegentlich ist es wünschenswert, eine sequentielle Auswertung von Formen auch außerhalb von speziellen Sprachkonstrukten explizit festlegen zu können. Dies ermöglichen die Systemfunktionen PROG1 und PROGN.
Ihre **Aufrufmuster** haben die Gestalt
(PROG1 <Form-1> <Form-2> ... <Form-n>)
und
(PROGN <Form-1> <Form-2> ... <Form-n>) mit $n \geq 0$

Auswertung:
Die Formen Form-1 bis Form-n werden in der gegebenen Reihenfolge nacheinander ausgewertet.

Wert der PROG1-Form ist der für die erste Form - also Form-1 - ermittelte Wert;

Wert der PROGN-Form ist der für die letzte Form - also Form-n - ermittelte Wert.

Mit Ausnahme der ersten bzw. letzten Form der Folge werden also alle Formen nur wegen ihrer Seiteneffekte ausgewertet.

Seiteneffekte:

Außer den mit der Auswertung von Form-1 bis Form-n verbundenen Seiteneffekten keine weiteren Seiteneffekte.

Beispiele:

Die Auswertung der folgenden Formen zeigt den Unterschied zwischen PROG1 und PROGN.

```
-->(PROG1 (SETQ A 'X) (SETQ B 'Y) (SETQ C 'Z))
X
-->(PROGN (SETQ A 'X) (SETQ B 'Y) (SETQ C 'Z))
Z
```

9.2 Map-Funktionen

Map-Funktionen ermöglichen eine Art der Iteration (= Wiederholung), bei der eine Funktion sukzessive auf Teile von einer oder mehreren Listen angewandt wird und dann die Resultate der einzelnen Funktionsanwendungen zu einer Liste zusammengefaßt werden.

Dabei gibt es verschiedene Möglichkeiten

- wie die Teile der Listen ausgewählt werden, auf welche die Funktion angewandt wird, und
- wie die Resultate der Funktionsanwendungen zu einer Liste zusammengefaßt werden.

Die wichtigsten Map-Funktionen sind zunächst MAPCAR und MAPLIST mit den

Aufrufmustern:

(MAPCAR <Funktionsbeschreibung> <-Liste-1-> ... <-Liste-n->)

und

(MAPLIST <Funktionsbeschreibung> <-Liste-1-> ... <-Liste-n->)

> Die Schreibweise <-Liste-1-> usw. bedeutet, daß jedes dieser Argumente als Wert eine Liste hat.
>
> Aus den Listen Liste-1 bis Liste-n werden die Argumente für die Auswertung der durch die Funktionsbeschreibung gegebenen Funktion gewonnen; deshalb muß die Anzahl dieser Listen auch dem "Argumentenbedarf" der Funktion entsprechen.

Auswertung bei MAPCAR:
Die gegebene Funktion wird zuerst mit den ersten Elementen (den CAR´s) der Listen ausgewertet, dann mit den zweiten Elementen (den CADR´s), dann mit den dritten Elementen (den CADDR´s), usw., bis eine der Listen erschöpft ist. Die Resultate dieser Auswertungen werden anschließend mit der Systemfunktion LIST zu einer Liste zusammengefaßt, die als Wert der MAPCAR-Form übergeben wird.

Beispiele:
```
-->(MAPCAR #'ABS '(3 -5 2 -6 -7))
(3 5 2 6 7)

-->(MAPCAR #'CONS '(A1 B1 C1) '((A2) (B2) (C2)))
((A1 A2) (B1 B2) (C1 C2))
```

Auswertung bei MAPLIST:
Die gegebene Funktion wird ausgewertet, wobei als Argumente zuerst die Listen selbst, dann die CDR´s dieser Listen, dann ihre CDDR´s usw. als Argumente dienen. Die Resultate dieser Auswertungen werden wieder mit der Systemfunktion LIST zu einer Liste zusammengefaßt, die als Wert der MAPLIST-Form übergeben wird.

Beispiele:
```
-->(MAPLIST #'(LAMBDA(X) (CONS 'A X)) '(B1 B2 B3 B4))
((A B1 B2 B3 B4) (A B2 B3 B4) (A B3 B4) (A B4))

-->(MAPLIST #'(LAMBDA(X) (COND ((MEMBER (CAR X) (CDR X)) 0)
                               (T 1)))
           '(A B A C D B C))
(0 0 1 0 1 1 1)          ; Der Resultatwert 1 zeigt an, daß das
                         ; in dieser Position stehende Element mindestens
                         ; noch ein zweites Mal in der Eingabeliste auftritt.
```

Seiteneffekte:
Sowohl bei MAPCAR wie bei MAPLIST entstehen außer den durch die Funktionsauswertungen selbst verursachten Seiteneffekten keine weiteren Seiteneffekte.

Im Idealfall haben die Listen Liste-1 bis Liste-n in den Aufrufen für MAPCAR und MAPLIST alle dieselbe Länge; ist dies nicht der Fall, so bricht die Iteration ab, sobald die kürzeste Liste aufgebraucht ist; überschüssige Elemente in den anderen Listen werden ignoriert.

Beispiel:

```
-->(MAPCAR #'CONS '(A1 B1 C1 D1 E1) '((A2) (B2) (C2)))
((A1 A2) (B1 B2) (C1 C2))
```

Von den **weiteren Map-Funktionen** erwähnen wir noch MAPC und MAPL; sie haben **Aufrufmuster** wie MAPCAR und MAPLIST und unterscheiden sich in der **Auswertung** von diesen allein dadurch, daß die Resultate der sukzessiven Funktionsanwendungen ignoriert werden. MAPC und MAPL werden benutzt, wenn die sukzessiven Funktionsaufrufe ausschließlich wegen ihrer Seiteneffekte ausgewertet werden. Als Wert wird von MAPC bzw. MAPL jeweils der Wert des zweiten Arguments der MAPC-Form bzw. MAPL-Form, also Liste-1 übergeben.

9.3 Allgemeine Iteration

Die im vorhergehenden Abschnitt behandelten Map-Funktionen ermöglichen eine spezielle Form der Iteration, bei der eine Funktion sukzessive auf Teile von einer oder mehreren Listen angewandt werden. Für die allgemeine Iteration steht in LISP die Systemfunktion DO zur Verfügung. Bemerkenswert an DO ist, daß von einem Iterationsschritt zum nächsten auch mehrere Iterationsvariable geändert werden können und zwar mit individuell festgelegten Schrittweiten.

Die Funktion DO hat das **Aufrufmuster:**

```
(DO   (  (<Variable-1> <Anfangswert-1> <Neuer-Wert-1>)
         (<Variable-2> <Anfangswert-2> <Neuer-Wert-2>)
         .
         .
         .
         (<Variable-n> <Anfangswert-n> <Neuer-Wert-n>) )
      (<Abbruchtest> <Resultatformen>)
      <Rumpf>)                                          mit n ≥ 1
```

Hier bedeuten Variable-1 bis Variable-n die sogenannten Iterationsvariablen.
Anfangswert-i und Neuer-Wert-i für $1 \leq i \leq n$ sind auswertbare Formen.
Abbruchtest ist eine Form.
Resultatformen steht für eine Folge von null oder mehr auswertbaren Formen.

Der Rumpf ist ein Programmstück mit null oder mehr auswertbaren Formen, unter denen auch eine spezielle Form mit dem Aufrufmuster (RETURN <Resultatform>) auftreten kann; dabei ist Resultatform eine auswertbare Form.

Auswertung:
Für den ersten Iterationsschritt werden zuerst alle Werte für Anfangswert-1 bis Anfangswert-n ermittelt und anschließend parallel den Iterationsvariablen zugewiesen. Das bedeutet, daß zum Zeitpunkt der Berechnung von Anfangswert-j noch keine der Iterationsvariablen Variable-i mit $1 \leq i < j$ an ihren Anfangswert-i gebunden ist und deshalb auch nicht verwendet werden kann. Ist keine Form für Anfangswert-i vorhanden, so tritt NIL als Ersatzwert ein. Dann wird der Abbruchtest ausgewertet. Falls er einen Wert ungleich NIL liefert, werden die als Resultatformen vorhandenen Formen ausgewertet und der Wert der letzten Form als Wert der DO-Form übergeben. Ist bei Resultatformen keine Form vorhanden, so wird NIL als Wert der DO-Form übergeben. Liefert der Abbruchtest den Wert NIL, so wird der Rumpf abgearbeitet. Falls dabei eine Form der Gestalt (RETURN <Resultatform>) ausgewertet wird, so wird die Abarbeitung der DO-Form damit abgeschlossen und der Wert der Resultatform als Wert der DO-Form übergeben, anderenfalls ist mit Abarbeitung des Rumpfes ein Iterationsschritt abgeschlossen. Für den nächsten Iterationsschritt werden zuerst die Werte für Neuer-Wert-1 bis Neuer-Wert-n ermittelt und anschließend parallel als neue Werte den Iterationsvariablen zugewiesen. Das bedeutet, daß während der Berechnung der neuen Werte alle Iterationsvariablen noch an ihre "alten" Werte gebunden sind und nur mit diesen genutzt werden können. Anschließend wird wieder der Abbruchtest ausgewertet und die Auswertung - wie oben beschrieben - weitergeführt.

Wir weisen darauf hin, daß bei der Spezifikation der Iterationsvariablen in DO-Formen entweder die Angabe des "Neuen Wertes" oder die Angaben von "Anfangswert" und "Neuem Wert" fehlen können.
Fehlt die Angabe zum "Neuen Wert", so wird für die Variable beim Übergang von einem Iterationsschritt zum nächsten kein neuer Wert berechnet. Fehlt auch die Angabe zum "Anfangswert", so wird hierfür NIL als Defaultwert genommen.

Beispiel:
Zur Berechnung der Länge einer Liste L hatten wir die rekursive Funktion OUR-LENGTH definiert:

```
(DEFUN OUR-LENGTH (L)
   (COND  ((NULL L) 0)
          (T (+ 1 (OUR-LENGTH (CDR L))))))
```

Die im folgenden definierte Version der Funktion OUR-LENGTH arbeitet iterativ:

```
(DEFUN OUR-LENGTH (L)
  (DO ((LST L (CDR LST))
       (COUNT 0 (+ 1 COUNT)))
      ((NULL LST) COUNT)))
```

Hier sind LST und COUNT Iterationsvariable; ihre Anfangswerte sind durch L und 0 gegeben. Mit jedem neuen Iterationsschritt werden ihre alten Werte LST und COUNT ersetzt durch (CDR LST) und (+ 1 COUNT).
Der Abbruchtest ist (NULL LST) ; es wird also abgebrochen, sobald LST die leere Liste ist. Wert der DO-Form wird der Wert von COUNT.
Der Rumpf der DO-Form ist leer.

Die Berechnung der neuen Werte der Iterationsvariablen könnte auch in den Rumpf der DO-Form verlegt werden; wir kämen dann zu folgender Formulierung:

```
(DEFUN OUR-LENGTH (L)
  (DO ((LST L)
       (COUNT 0))
      ((NULL LST) COUNT)
      (SETQ LST (CDR LST))
      (SETQ COUNT (+ 1 COUNT))))
```

9.4 Rekursion versus Iteration: ein Beispiel

Die Umkehrung einer Liste kann - wie wir wissen - mit der Systemfunktion REVERSE ermittelt werden. Im folgenden wollen wir verschiedene rekursive und iterative Definitionen für eine Funktion zur Umkehrung einer Liste erarbeiten. Dies illustriert die Unterschiede zwischen rekursiven und iterativen Lösungen.

Eine rekursive Lösung ist dadurch charakterisiert, daß die Bearbeitung der gegebenen Aufgabe auf die Bearbeitung einer Aufgabe derselben Art aber mit "einfacheren" Daten zurückgeführt wird: Offensichtlich erhält man die Umkehrung einer Liste wie (A B C D), indem man die Umkehrung der Liste (B C D) mit der Liste (A) in dieser Reihenfolge durch APPEND zusammenfügt. Dabei ist nur noch eine Liste mit 3 Elementen umzukehren. Dies wird entsprechend

auf die Umkehrung einer Liste mit 2 Elementen zurückgeführt. So fährt man fort, bis nur noch die leere Liste umzukehren ist. Deren Umkehrung ist aber wieder die leere Liste. Damit kommen wir zu folgender Definition:

```
(DEFUN OUR-REVERSE (LST)
   (COND    ((NULL LST) 'NIL)
            (T (APPEND (OUR-REVERSE (CDR LST))
                       (CONS (CAR LST) 'NIL)))))
```

Diese Funktion OUR-REVERSE leistet die Umkehrung einer Liste, arbeitet aber sehr ineffizient, da nach jedem rekursiven Aufruf von REVERSE das Resultat an die rekursive Funktion APPEND übergeben wird, die es in einer Folge von CAR- und CDR-Aufrufen wieder zerlegt.
Man kann nun versuchen, bei der rekursiven Rückführung der Aufgabe anders vorzugehen:
Um die Umkehrung von (A B C D) zu bestimmen, "CONSen" wir zunächst A an eine leere Liste an und fügen dann sukzessive den Listenkopf der jeweils noch verbleibenden umzukehrenden Liste vorne an. Dies führt auf folgende Definition:

```
(DEFUN R-REVERSE (LST)
   (R-AUX-REVERSE LST 'NIL))

(DEFUN R-AUX-REVERSE (REM AUX)
   (COND    ((NULL REM) AUX)
            (T (R-AUX-REVERSE (CDR REM) (CONS (CAR REM) AUX)))))
```

Im Rumpf von R-REVERSE wird die rekursive Hilfsfunktion R-AUX-REVERSE aufgerufen. Diese hat zwei Parameter: REM enthält zu jedem Zeitpunkt denjenigen Rest der ursprünglichen Liste, der noch zu invertieren ist; AUX enthält die invertierte Liste, soweit sie bis dahin aufgebaut ist. Wenn (NULL REM) den Wert T hat, also von der ursprünglichen Liste nichts mehr übrig ist, enthält AUX die ganze invertierte Liste und gibt das Resultat an. Andernfalls wird R-AUX-REVERSE rekursiv aufgerufen mit neuen Parameterwerten, die sich ergeben, indem man aus REM, dem Rest der ursprünglichen Liste, das erste Element entfernt und als neuen Listenkopf an AUX den bereits invertierten Teil "anCONSt".

Diese Definition einer Funktion zur Umkehrung einer Liste arbeitet aber im Prinzip schon iterativ. Dies legt eine Überführung in eine iterative Version nahe. Wir kommen damit zu

```
(DEFUN OUR-REVERSE (LST)
  (DO ((REM LST (CDR REM))
       (AUX 'NIL (CONS (CAR REM) AUX)))
       ((NULL REM) AUX)
          ;          Rumpf der DO-Form ist leer
       ))
```

Dieses iterative und das obige rekursive Programm arbeiten praktisch in gleicher Weise:
Die Variablen REM und AUX fungierten zunächst als Parameter einer rekursiven Funktion, dann als Iterationsvariable, haben aber beide Male dieselbe Aufgabe. Sie übernehmen in beiden Programmen dieselben Anfangswerte LST bzw. NIL und werden bei Rekursionsschritten und Iterationsschritten in gleicher Weise verändert:
REM gibt das erste Listenelement ab; dieses wird an AUX "angeCONSt". Sobald REM leer ist, gibt AUX das Resultat.

9.5 Zusammenfassung

Kontrollstrukturen spielen bei imperativen Programmiersprachen eine zentrale Rolle; ihre Bedeutung bei funktionalen Sprachen beruht im wesentlichen auf dem Vorhandensein von Seiteneffekten.
Die Formen PROG1 und PROGN bewirken eine sequentielle Auswertung von Formen und übergeben als Wert den Wert der ersten bzw. letzten Form.

Map-Funktionen ermöglichen eine besondere Form der Iteration, bei der eine Funktion sukzessive auf bestimmte Teile von einer oder mehreren Listen angewendet wird und die Resultate der sukzessiven Funktionsanwendungen in vorgegebener Weise zu Listen zusammengefaßt werden.
Die DO-Form ermöglicht eine sehr allgemeine und flexible Iteration, die kaum noch Wünsche offen läßt.

KAPITEL 10:

Grundlagen der Ein- und Ausgabe

In Kapitel 2 wurde der Interpretationszyklus eines LISP-Systems beschrieben. Danach liest ein LISP-Interpretierer jeweils eine Form vom Terminal ein, evaluiert sie unter den gegebenen Bindungen, gibt den erhaltenen Wert aus und beginnt diesen Zyklus von Neuem.
Im Prinzip genügen die im Interpretationszyklus implizit enthaltenen Ein- und Ausgabeanweisungen für die Kommunikation zwischen LISP-System und Benutzer.

In der Praxis sollte man jedoch verschiedene Ein- und Ausgabefunktionen explizit aufrufen können, welche die Entwicklung, Haltung und Abarbeitung umfangreicher Programme, die Eingabe und Haltung großer Datenmengen sowie die Erzeugung gut lesbarer Ausgabedaten erleichtern.

Ziel dieses Kapitels ist es, die Grundlagen für Ein- und Ausgabe aufzuzeigen und Werkzeuge vorzustellen, die bereits eine recht vernünftige Steuerung der Ein- und Ausgabe ermöglichen.

Die Möglichkeiten der Ein- und Ausgabe in COMMON LISP können hier auch nicht annähernd erschöpfend dargelegt werden.

10.1 Ströme als Kommunikationswege

Um umfangreichere Bestände von Funktionsdefinitionen und Daten aufbauen und für eine spätere Wiederverwendung aufbewahren zu können, werden Dateien (engl. "Files") angelegt und Kommunikationswege zwischen LISP-System und Außenwelt einschließlich File-System aufgebaut. Diese Kommunikationswege nennen wir Ströme (engl. "Streams").
Beispiele für Ströme sind die Kommunikationswege für
die Standardeingabe (i.a. über die Tastatur),
die Standardausgabe und
die Standardfehleranzeige (beide i.a. über den Bildschirm).

Jedes LISP-System verfügt über verschiedene globale Variable, deren Werte Ströme sind.

Zu den wichtigsten gehören

STANDARD-INPUT
STANDARD-OUTPUT
ERROR-OUTPUT

Ihre Werte geben jeweils diejenigen Ströme an, über die die Standard-Eingabe, die Standard-Ausgabe sowie die Fehlermeldungen laufen.

Ströme als Kommunikationswege für Daten müssen erzeugt und nach Verwendung wieder geschlossen werden.
Die **Auswertung** der Form

(OPEN <Filename> :DIRECTION :INPUT)

bewirkt als **Seiteneffekt** die Eröffnung eines Stroms vom angegebenen File in das LISP-System. Die Angabe von :DIRECTION und :INPUT in der Position des 2. und 3. Arguments legt die Richtung der Kommunikation fest: Der eröffnete Strom ist also ein Eingabestrom.

Wert der OPEN-Form ist dieser Eingabestrom.

Durch eine LET-Bindung wie in

```
(LET    ((STREAM (OPEN <Filename> :DIRECTION :INPUT)))
        .
        <Rumpf der LET-Form>
        .
                )
```

kann eine Variable, etwa STREAM, an einen Eingabestrom als Wert gebunden werden.
Dieser Eingabestrom kann durch Auswertung von

(CLOSE STREAM)

wieder geschlossen werden. Dies beruht auf dem **Seiteneffekt** der CLOSE-Form.

Wert einer CLOSE-Form ist grundsätzlich T.

Sobald ein Strom geschlossen ist, können keine Ein- oder Ausgabeoperationen mehr über ihn abgewickelt werden.

Da Programme beim Entwurf selten auf Anhieb fehlerfrei sind, erlebt man nicht allzuviel Freude, wenn man sie gleich in den Interpretationszyklus eingibt.

Zweckmäßig schreibt man sie zunächst in ein File. Wie das geschehen kann, hängt weitgehend vom jeweiligen Betriebssystem ab und ist nicht Gegenstand der Sprachdefinition von LISP. Die in einem File gespeicherten LISP-Formen müssen zur Auswertung in das LISP-System geladen werden. Dies leistet ein Aufruf der Funktion LOAD mit dem
Aufrufmuster:

(LOAD <Filename>)

Ihre **Auswertung** bewirkt als **Seiteneffekt** ein Laden aller Ausdrücke aus dem File in das LISP-System und deren Auswertung in der gegebenen Reihenfolge.

Bei erfolgreichem Laden eines Files liefert LOAD einen **Wert** ungleich NIL; dieser Wert ist systemabhängig und nicht durch die Sprachdefinition festgelegt.

LOAD hat noch sogenannte **optionale** Argumente, das sind Argumente, die nicht unbedingt angegeben werden müssen. Durch Angabe dieser Argumente kann man festlegen, daß über den Ladevorgang Meldungen ausgegeben werden, daß die Werte der ausgewerteten Formen über die Standardausgabe ausgegeben werden, oder daß das System bestimmte Meldungen absetzt, wenn das zu ladende File nicht existiert. Wir gehen darauf hier nicht näher ein.

10.2 Einlesen mit READ

Die wichtigste Funktion zur Eingabe von Ausdrücken durch Einlesen ist READ. Sie hat das
Aufrufmuster:

(READ <Strom> <Fehlermeldung am File-Ende> <Default-Wert am File-Ende>)

Alle 3 Argumente sind optional, müssen also beim Aufruf von READ nicht unbedingt angegeben werden. Zur Auswertung werden die jeweils vorhandenen Argumente den Parametern der Reihe nach zugeordnet. Für Parameter, die bei dieser Zuordnung leer ausgehen, wird der zugehörige Default-Wert zugrundegelegt. Das bedeutet, daß einem Parameter im Funktionsaufruf nur dann ein

Argument mitgegeben werden kann, wenn alle in der Parameterliste voranstehenden Parameter ebenfalls mit Argumenten versorgt werden.

Zur Auswertung

eines Aufrufs von READ werden nun als **Seiteneffekt** solange Zeichen von dem durch das 1. Argument bezeichneten Eingabestrom gelesen bis ein vollständiger Ausdruck eingelesen ist. Dieser wird dann auch **Wert** der Form.

Der Default-Wert für Strom ist T; er bezeichnet die Standardeingabe.

Wird beim Einlesen das File-Ende erreicht, bevor ein vollständiger Ausdruck eingelesen ist, weil z.B. nicht genügend Klammern da sind, um den Ausdruck korrekt abzuschließen, so wird ein Fehler gemeldet.

Übrigens kann beim Lesen von der Standardeingabe das File-Ende nicht erreicht werden. Hier wartet das System grundsätzlich bis der Benutzer einen vollständigen Ausdruck eingegeben hat und stellt solange alle Aktivitäten ein.

Frage:

Was passiert, wenn mit dem Lesen eines Ausdrucks von einem File gerade das File-Ende erreicht wurde und daraufhin erneut ein READ-Aufruf erfolgt ?

In diesem Fall wird nur dann ein Fehler gemeldet, wenn das 2. Argument einen Wert ungleich NIL hat; hat es den Wert NIL, so wird der Wert des 3. Arguments als Ersatz für einen gelesenen Ausdruck herangezogen. Der Default-Wert für das 2. Argument ist T, er veranlaßt somit eine Fehlermeldung.

Beispiel:

Wir tippen auf das Bereitschaftszeichen des Systems hin die Zeichenfolge (READ) ein. Das ist ein Aufruf von READ ohne Angaben für die optionalen Argumente. Der Default-Wert für das erste Argument legt fest, daß die Eingabe über den Standard-Eingabestrom - also im allgemeinen über die Tastatur - kommt. In diesem Fall sind auch die Default-Werte für die weiteren Argumente irrelevant. Das System stellt alle Aktivitäten ein und wartet darauf, daß der Benutzer am Standard-Eingabegerät einen Ausdruck eingibt. Dieser Ausdruck wird eingelesen und bildet ohne weitere Auswertung den Wert der Form (READ).

Also

-->(READ) DREINGABE
DREINGABE

Wir sind nun in der Lage, eine eigene Version für die Systemfunktion LOAD zu schreiben

```
(DEFUN OUR-LOAD (FILE-NAME)
    (LET ((STREAM (OPEN FILE-NAME :DIRECTION :INPUT)))
         (DO   ((FORM NIL (READ STREAM NIL STREAM)))
               ((EQ FORM STREAM) (CLOSE STREAM))
               (EVAL FORM))))
```

OUR-LOAD hat einen Parameter FILE-NAME für das File, das geladen werden soll. Der Rumpf von OUR-LOAD besteht aus einer LET-Form. Diese hat eine lokale Variable, nämlich STREAM; sie wird gebunden an den durch die OPEN-Form eröffneten Strom, der das durch FILE-NAME bezeichnete File mit dem LISP-System verbindet und zwar für Eingabe aus dem File in das LISP-System. Der Rumpf der LET-Form besteht aus einer DO-Form. Die DO-Form verwendet genau eine Laufvariable, nämlich FORM. Ihr Anfangswert ist NIL. Den neuen Wert erhält sie jeweils durch Lesen eines Ausdrucks von dem durch STREAM bezeichneten Eingabestrom. So erhält FORM als neuen Wert immer den nächsten Ausdruck aus dem File, der dann durch Aufruf von EVAL ausgewertet wird.

Da alle Ausdrücke vom File gelesen und geladen werden sollen, aber ihre Anzahl nicht bekannt ist, müssen wir das File-Ende erkennen können und sobald es erkannt ist, die DO-Schleife abbrechen. Um das File-Ende erkennen zu können, wird in den wiederholten Aufrufen von READ für das zweite Argument der Wert NIL und für das dritte Argument, das den "Ersatzwert" festlegt, ein besonderer Wert gewählt, der mit Sicherheit im File nicht auftritt. Sobald die Auswertung von READ diesen Wert liefert, ist das File-Ende erreicht.
Einen besonderen Wert anzugegben, der mit Sicherheit nicht auf dem File steht, ist einfach. Der Wert von STREAM ist der Strom vom File in das LISP-System und dieser kann nicht als Text gedruckt oder von außen eingelesen werden. Er wurde oben im dritten Argument von READ eingesetzt.

Die Funktion OUR-LOAD arbeitet wie die System-Funktion LOAD.

Ein Aufruf bewirkt als **Seiteneffekt** Laden und Auswerten aller Formen des angesprochenen Files, und liefert als **Wert** den Wert der letzten Form der Testklausel

((EQ FORM STREAM) (CLOSE STREAM)),

also den Wert von (CLOSE STREAM). Dieser aber ist T.

Sollen die Werte der ausgewerteten Formen ausgegeben werden, so muß OUR-LOAD um eine Anweisung zur Ausgabe des Wertes von (EVAL FORM) ergänzt werden. Dazu braucht man jedoch nur

(EVAL FORM) durch (PRINT (EVAL FORM))

zu ersetzen.
Damit haben wir allerdings schon dem nächsten Abschnitt vorgegriffen, in dem wir uns mit Ausgabefunktionen beschäftigen.

10.3 Ausgabe mit PRINT

Die Möglichkeit der Ausgabe mit Hilfe der Systemfunktion PRINT wurde bereits erwähnt. Diese kann aufgerufen werden mit dem
Aufrufmuster:

(PRINT <Form> <Strom>)

Das erste Argument bestimmt, was ausgegeben wird.
Das zweite Argument ist optional; es bestimmt über welchen Strom, also wohin ausgegeben wird. Sein Default-Wert ist T. Welchen Strom dieser Default-Wert spezifiziert, ist durch den Wert der globalen Variablen *STANDARD-OUTPUT* festgelegt. Im Normalfall ist das der Bildschirm des Terminals.

Zur **Auswertung**
der obigen Print-Form wird der Wert der Form ermittelt, dann wird als **Seiteneffekt** eine Ausgabe über den spezifizierten Ausgabestrom durchgeführt. Dort wird zunächst ein Übergang auf eine neue Zeile vorgenommen, dann der Wert der Form ausgegeben und an diesen noch ein Leerzeichen angefügt.

Wert der PRINT-Form ist der **Wert** des ersten Arguments; er wird wie üblich über den Bildschirm ausgegeben.

Ist durch den Ausgabestrom das Bildschirmterminal als Ausgabemedium festgelegt, so erscheint die **Ausgabe** auf diesem **doppelt**. Zuerst aufgrund des Seiteneffekts und dann als Wert der Form.

Beispiele:

```
-->(PRINT (+ 17 4) 'T)                    ; Ausgabe auf das Terminal
21                                        ; aufgrund des Seiteneffekts
21                                        ; als Wert der Form

-->(PRINT "NA SOWAS" 'T)                  ; erstes Argument ist ein String
"NA SOWAS"                                ; dieser hat sich selbst als Wert
"NA SOWAS"
```

Beispiel:

Bei Aufruf der folgenden Funktion werden in einer Endlos-Schleife Quadratzahlen ausgedruckt:

```
-->(DEFUN QUADRATE ( ))                   ; leere Parameterliste
     (DO   ((N 0 (+ N 1)))
           (NIL)                          ; Abbruchtest
           (PRINT (* N N))))
QUADRATE

-->(QUADRATE)
0
1
4
9
16                                        ; diese DO-Schleife bricht nicht ab
.
.
.
```

So ziemlich jedes LISP-System bietet dem Benutzer eine Möglichkeit, Berechnungen, die anscheinend in eine Endlos-Schleife geraten sind, abzubrechen. Es ist zweckmäßig sich darüber frühzeitig zu informieren.

Beispiel:

Ein File enthält einen Datensatz, mit mindestens 144 Zahlen, die aufsummiert werden sollen. Für die Überprüfung der Anzahl und die Summation der Zahlen kann folgende Funktion verwendet werden:

```
(DEFUN SUMME-DATENSATZ (FILENAME)
    (LET  ((STREAM (OPEN FILENAME :DIRECTION :INPUT)))
          (DO  ((SUMME 0.0 (+ (READ STREAM 'T) SUMME))
                (I 144 (- I 1)))
               ((ZEROP I)
                  (COND ((EQ (STREAM (READ STREAM NIL STREAM)))
                         (CLOSE STREAM))
                        (T (PRINT "ZUVIELE EINTRÄGE")))
                  SUMME))))
```

Die Funktion SUMME-DATENSATZ hat einen Parameter FILENAME für das File, das den Datzensatz enthält. Ihr Rumpf besteht aus einer LET-Form. Diese hat eine lokale Variable, nämlich STREAM; diese wird gebunden an den durch die OPEN-Form eröffneten Strom, der das durch FILENAME bezeichnete File mit dem LISP-System verbindet und zwar für Eingabe aus dem File in das LISP-System. Der Rumpf der LET-Form besteht aus einer DO-Form; die DO-Form enthält 2 Laufvariable, nämlich SUMME und I. Ihre **Anfangswerte** sind 0.0 bzw. 144. Die neuen Werte erhält man bei SUMME durch Lesen der nächsten Zahl vom File und Addition zum bisherigen Wert von SUMME und bei I durch Subtraktion von 1. Der Abbruchtest der DO-Form gibt den Wert T, wenn (ZEROP I) den Wert T, also I den Wert 0 hat. Nur für I gleich 0 werden die zugehörigen Resultatformen abgearbeitet, wobei die letzte - das ist hier SUMME - den Wert der DO-Form ergibt. Die erste Resultatform ist eine COND-Form. Im Test ihrer ersten Klausel wird überprüft, ob das File-Ende erreicht ist. Diese Technik der Überprüfung ist uns bekannt. Hat das zweite Argument von READ den Wert NIL, so dient der Wert des dritten Arguments als Ersatz, wenn das File-Ende erreicht ist und eigentlich nichts mehr gelesen werden kann. Dabei muß sicher sein, daß der Wert des dritten Arguments nicht im File steht. Der Wert von STREAM erfüllt diese Bedingung.

Ist die Bedingung der ersten Klausel nicht erfüllt, so enthält das File mehr als 144 Einträge. Eine entsprechende Ausgabe wird in der zweiten Klausel veranlaßt.

Die zweite Resultatform, nämlich SUMME, gibt bei Abbruch den Wert der DO-Form an.

Was geschieht aber jetzt, wenn das File zu wenig Summanden enthält ?

Dies wird erkannt bei Auswertung der READ-Form zur Ermittlung des "neuen Wertes" von SUMME. Die Form (READ STREAM 'T) bewirkt ein Lesen des nächsten Wertes über den mit dem File verbundenen Eingabestrom. Das zweite Argument von READ hat einen Wert ungleich NIL. Wie wir oben bei READ gelernt haben, bewirkt dies, daß eine Fehlermeldung ausgegeben wird, wenn READ aufgerufen wird, obwohl das File-Ende schon erreicht ist.

10.4 Exotische Symbolbezeichnungen

Gelegentlich kann es sinnvoll sein, Symbolbezeichnungen zu verwenden, die eigentlich nach den Regeln (der Syntax) nicht zugelassen sind, zum Beispiel ein Symbol, das nur aus der öffnenden Klammer oder aus zwei Zeichenfolgen mit einem Leerzeichen dazwischen bestehen soll.
Hier helfen sogenannte Fluchtsymbole wie | oder \ . Der senkrechte Strich | dient als vielfaches Fluchtsymbol. Das heißt, eine Folge von beliebig vielen Zeichen, die damit eingerahmt ist, verliert ihre eventuell sonst vorhandenen syntaktischen Eigenschaften.
Der Schrägstrich \ ist ein einfaches Fluchtsymbol; das heißt, sein Auftreten zeigt an, daß nur das nachfolgende Zeichen seine syntaktischen Eigenschaften verliert.
In der Ausgabe verwendet LISP allerdings grundsätzlich das Symbol | als "klammerndes" Fluchtsymbol. Dabei wird das abschließende | stets soweit wie möglich nach rechts verschoben.

Beispiele:

```
-->(SETQ SYMBOL1 '|) (|)
|) (|

-->(SETQ SYMBOL2 '\))
|) |                                   ; Ausgabe mit Fluchtsymbol |

-->(SETQ SYMBOL3 '|DUMMES ZEUG|)
|DUMMES ZEUG|

-->(SETQ SYBMOL4 '\(K \)ARL)
|(K)ARL|                               ; Verschiebung des abschließenden
                                      ; Fluchtsymbols | nach rechts

-->(PRINT SYMBOL1)
|) (|                                  ; erste Ausgabe aufgrund des
|) (|                                  ; Seiteneffekts der Funktion PRINT
                                      ; zweite Ausgabe als Wert der
                                      ; PRINT-Form

-->(PRINT SYMBOL2)
|) (|
|) (|
```

```
-->(PRINT SYMBOL3)
|DUMMES ZEUG|
|DUMMES ZEUG|

-->(PRINT SYMBOL4)
|(K)ARL|
|(K)ARL|
```

10.5 Einfache Variationen von PRINT

Im folgenden sei noch auf zwei Variationen von PRINT hingewiesen, welche die Erstellung einer gefälligen Ausgabe erleichtern können.
Sie haben beide dasselbe **Aufrufmuster wie PRINT**, nämlich:

(PRIN1 <Form> <Strom>)
(PRINC <Form> <Strom>)

Als **Seiteneffekt** wird jeweils der Wert des 1. Arguments über den durch das 2. Argument bezeichneten Ausgabestrom ausgegeben. Wie bei PRINT ist das 2. Argument jeweils optional; falls es fehlt, wird entsprechend dem Default-Wert T über die Standard-Ausgabe ausgegeben.

Wert der Form ist jeweils der Wert des 1. Arguments.

Für die Ausgabe aufgrund des Seiteneffekts, nicht aber für die Ausgabe als Wert der Form, gilt:

PRIN1 unterscheidet sich in der Wirkung von PRINT allein dadurch, daß der Zeilenvorschub vor der Ausgabe und das Leerzeichen nach der Ausgabe fehlen.

PRINC unterscheidet sich in der Wirkung von PRIN1 dadurch, daß Fluchtsymbole und die doppelten Anführungszeichen bei Strings (vorne und hinten) nicht ausgegeben werden.
Auch für andere Datentypen kann PRINC Unterschiede in der Ausgabe aufweisen. Die allgemeine Regel ist, daß PRINC eine schön aussehende Ausgabe liefern soll, während Ausgabe durch PRIN1 grundsätzlich auch als Eingabe für READ akzeptierbar sein muß.

Beispiel:

```
-->(SETQ L  '("L" IST \(K \)EINE
              |LEE( )RE| LIS( )TE))
("L" IST  | (K)EINE |  |LEE( )RE|  LIS NIL TE)
```

Jetzt wenden wir mittels MAPCAR die drei PRINT-Funktionen PRINT, PRIN1 und PRINC auf die Elemente von L an.
Wir leiten auch die Ausgabe aufgrund des Seiteneffekts auf den Bildschirm, indem wir für den optionalen Parameter Strom keinen Wert angeben, so daß der Default-Wert zugrundegelegt wird. Damit erscheint die Ausgabe auf dem Bildschirm jeweils zweimal, zuerst aufgrund des Seiteneffekts und danach als Wert der Form.

```
-->(MAPCAR #'PRINT L)
"L"          ;
IST          ;
| (K)EINE |  ;
|LEE( )RE|   ,        Ausgabe aufgrund des Seiteneffekts
LIS          ;
NIL          ;
TE           ;
("L" IST |(K)EINE| |LEE( )RE| LIS NIL TE)    ; Ausgabe als Wert der Form

-->(MAPCAR #'PRIN1 L)
("L"IST|(K)EINE||LEE( )RE|LISNILTE)
("L" IST |(K)EINE| |LEE( )RE| LIS NIL TE)

-->(MAPCAR #'PRINC L)
LIST(K)EINELEE( )RELISNILTE
("L" IST |(K)EINE| |LEE( )RE| LIS NIL TE)
```

Man erkennt, daß der **Seiteneffekt** der PRIN1- und PRINC-Anweisung die Ausgabe jeweils in derselben Zeile veranlaßt, im Falle der PRINC-Anweisung allerdings mit Unterdrückung des Fluchtsymbols |. Die zweite Ausgabe gibt jeweils den **Wert** der zugehörigen Form.

Die Funktion TERPRI (von **Ter**minate **Pr**inting) mit dem
Aufrufmuster

(TERPRI <Strom>)

bewirkt in der Ausgabe auf dem angegebenen Ausgabestrom einen Übergang auf eine neue Zeile. Das Argument Strom ist optional. Falls es fehlt, wird entsprechend dem Defaultwert T auf der Standard-Ausgabe ein Zeilenvorschub vorgenommen.

Wert der TERPRI-Form ist NIL.

Offensichtlich hat die Auswertung von
(PRINT <Form>)

denselben Seiteneffekt wie die Auswertung von

(PROGN (TERPRI) (PRIN1 <Form>) (PRINC '| |))

führt aber zu einem anderen Wert, nämlich | |.

10.6 Zusammenfassung

Wir haben in dieser Lektion einige einfachere Möglichkeiten der Ein- und Ausgabe in COMMON LISP kennengelernt. Diese ermöglichen bereits eine recht vernünftige Steuerung der Ein- und Ausgabe, und reichen auch dann, wenn eine umfangreichere Teilmenge von LISP benutzt wird, als bisher vorgestellt wurde.

Zum Standard von COMMON LISP gehört noch ein reiches Spektrum von Funktionen zum Datentransfer auf Strömen, zur Kommunikation zwischen File-System und LISP-System sowie zur unmittelbaren Ein- und Ausgabe. Für formatierte Ausgabe dient insbesondere die Systemfunktion FORMAT (siehe Kapitel 13); sie ermöglicht eine Steuerung der Ausgabe, die eigentlich keine Wünsche offen läßt.

Wir haben jetzt am Ende von Teil I dieser Einführung die grundlegenden Eigenschaften und die wichtigsten Sprachelementen von LISP kennengelernt. Damit können im Prinzip alle Probleme bearbeitet werden, die überhaupt algorithmisch lösbar sind. Bei Anwendungen entsteht jedoch der Wunsch nach besser angepaßten Daten- und Programmstrukturen, kurz: nach mehr Komfort. Diesen Komfort wollen wir im anschließenden Teil II kennenlernen.

Teil II

Weiterer Ausbau

KAPITEL 11:

Listen

(noch einmal, weil sie so wichtig sind)

Die grundlegende Datenstruktur in LISP sind Listen. Die Bezeichnung LISP als Akronym für "LIST Processing" unterstreicht dies deutlich. Da LISP-Programme auch selbst durch Listen dargestellt und wie Listen verarbeitet werden können, verschwindet der Unterschied zwischen Daten und Programmen.

Wir wollen in diesem Kapitel unser Wissen über die Verwendung von Listen als Datenstrukturen vertiefen.
Zunächst gehen wir auf die Korrespondenz zwischen der externen Darstellung von Listen und der zugrundeliegenden internen Darstellung mit Hilfe von binären Bäumen ein.
Anschließend behandeln wir Listenoperationen für zwei wichtige Anwendungsbereiche, nämlich

- Darstellung und Manipulation von Mengen
  sowie
- Darstellung und Manipulation von Assoziationen.

Zum Abschluß betrachten wir verschiedene Funktionen zum Kopieren von Listen bzw. der ihnen zugrundeliegenden Speicherstrukturen.

Ziel dieses Kapitels ist neben der Vermittlung neuen Stoffes vor allem auch die Festigung und Vertiefung unseres bisherigen Wissens über Listen.

11.1 Listen und Bäume

Die Listen der Sprache LISP werden allgemein rekursiv definiert:

Eine Liste ist	entweder	die leere Liste, bezeichnet mit ( ) oder mit NIL
	oder	entsteht durch Anwendung von CONS auf 2 Argumente, deren Werte Atome oder Listen sind.

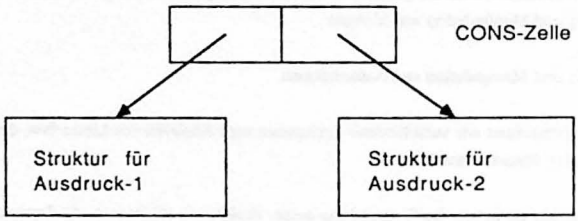
Diese Definition charakterisiert gerade die sogenannten Punkt-Listen. Sie zeigt aber auch einen häufig begangenen Weg bei der rekursiven Lösung einer Aufgabe:

Eine Aufgabe für Listen kann häufig gelöst werden, indem man dieselbe Aufgaben für die Bestandteile löst, aus denen die ursprüngliche Liste aufgebaut ist. So führt man kompliziertere Aufgaben für kompliziertere Listen zurück auf einfachere und schließlich triviale Aufgaben.

Betrachten wir nun den Aufbau von Listen aus einfacheren Listen: Die Aufbaufunktion CONS verwendet als Bausteine Atome oder Listen und baut daraus umfangreichere Listen auf. Jeder Aufbauschritt wird durch eine Anwendung von CONS beschrieben. Dabei werden intern die Speicherstrukturen für die Bausteine miteinander verknüpft durch Einrichtung einer CONS-Zelle und entsprechende Zeigerstrukturen:
Jeder Anwendung von CONS auf zwei Argumente Ausdruck-1 und Ausdruck-2, also jedem Aufruf der Gestalt

(CONS <Ausdruck-1> <Ausdruck-2>)

entspricht intern die Einrichtung einer CONS-Zelle als Knoten einer Verzweigung zu den Strukturen für die Argumente Ausdruck-1 und Ausdruck-2.



Betrachtet man die rekursive Definition von Listen und die Realisierung eines Aufbauschrittes im Speicher, so erkennt man: Die Liste als grundlegende Datenstruktur von LISP ist letztlich nur ein Mittel zur Darstellung von binären Bäumen. Die eigentlich zentrale Datenstruktur im Speicher sind **binäre Bäume.**

Auf folgende Details sei noch besonders hingewiesen:

In der oben gegebenen Definition von Listen sind in den Anwendungen der Funktion CONS für beide Argumente als Werte Atome und allgemeine Listen (also Punkt-Listen) zugelassen. Läßt man bei CONS im zweiten Argument als Wert nur Listen im engeren Sinne zu, so erhält man eine

Charakterisierung der sogenannten echten Listen; wir sprechen auch von normalen Listen. Sie bilden eine echte Teilmenge der allgemeinen Punkt-Listen und ermöglichen eine Vereinfachung in der Notation. Die kleine Unsymmetrie in den Argumenten bei der Anwendung von CONS spiegelt sich natürlich auch in der Darstellung der Listen durch binäre Bäume wieder.

11.2 Mengen

Eine Liste kann auch als Darstellung der Menge ihrer Elemente (auf dem obersten Niveau) angesehen werden. Es gibt dazu eine Reihe von Systemfunktionen, die bei dieser Interpretation von Listen gerade die üblichen Mengenoperationen beschreiben. Eine der wichtigsten sind sicherlich **MEMBER** und **ADJOIN** . MEMBER überprüft, ob ein Objekt Element einer Liste bzw. der dargestellten Menge ist (siehe Kapitel 4). ADJOIN führt diese Überprüfung ebenfalls durch und fügt das Objekt, sofern es noch nicht vorhanden ist, als neues Element ein (siehe Kapitel 6).

Mit den Funktionen **UNION** und **INTERSECTION** können Vereinigung und Durchschnitt von zwei Listen ermittelt werden. Sie haben die
Aufrufmuster:

(UNION <-Liste-1-> <-Liste-2->)
(INTERSECTION <-Liste-1-> <-Liste-2->)

Die Werte der Argumente sind also jeweils Listen.

Da alle Mengen mit Listen dargestellt werden, sind einige Besonderheiten zu beachten.

Wert von UNION
ist eine Liste all derjenigen Elemente, die in einer der beiden Listen oder in beiden Listen auf dem obersten Niveau erscheinen.
Die Reihenfolge der Elemente in der Resultatliste ist jedoch nicht festgelegt. Enthält eine der beiden Argumentlisten Duplikate, so können diese auch in der Resultatliste dupliziert auftreten.

Wert von INTERSECTION
ist eine Liste all derjenigen Elemente, die in jeder der beiden Listen auf dem obersten Niveau erscheinen.
Auch hier ist die Reihenfolge der Elemente in der Resultatliste nicht festgelegt. Enthalten beide Argumentlisten Duplikate ein- und desselben Elements, so können diese Duplikate in der Resultatliste ebenfalls auftreten, müssen aber nicht.

Beispiele:

```
-->(UNION '(A B C) '(D A F))
(A B C D F) oder (B C D A F) oder (C D B A F) oder .....

-->(INTERSECTION '(A B C) '(D A F))
(A)
```

UNION und INTERSECTION verwenden beide EQL als Vergleichsprädikat, falls nichts anderes festgelegt ist. Die Festlegung eines anderen Testprädikats wird ähnlich wie bei MEMBER vorgenommen. Wir kommen darauf in Kapitel 14 zurück. Vom zugrundegelegten Vergleichsprädikat hängt übrigens auch ab, ob bestimmte Elemente als Duplikate angesehen werden oder nicht.

Das Vergleichsprädikat EQL überprüft bei Listenstrukturen die Gleichheit der Zeiger. Dies kann überraschende Konsequenzen haben:

Beispiel:

```
-->(INTERSECTION '(A () (S) D) '((S) D B ()))
(D NIL)                          ; den beiden Auftreten von (S) liegen
                                 ; verschiedene Speicherstrukturen zugrunde:
                                 ; die Liste (S) wird nicht als gemeinsames
                                 ; Listenelement erkannt.
```

11.3 Assoziationslisten

Eine Assoziationsliste, kurz A-Liste, ist eine in LISP häufig benutzte Datenstruktur. Eine A-Liste ist eine Liste von CONS-Paaren; jedes Paar beschreibt eine Assoziation. Die erste Komponente (der CAR-Teil) einer solchen Assoziation ist ein Symbol, der sogenannte **Schlüssel** (key); die zweite Komponente (der CDR-Teil) ist ein dem Schlüssel zugeordneter Wert; wir sprechen auch vom zugeordneten **Datum**. A-Listen können aufgebaut werden mit den Systemfunktionen ACONS und PAIRLIS.

ACONS fügt ein Assoziationspaar an eine bestehende A-Liste an. Sie hat das

Aufrufmuster:

(ACONS <Schlüssel> <Datum> <-A-Liste->)

Das dritte Argument muß als Wert eine A-Liste haben.

Auswertung:
ACONS bildet aus den ersten beiden Argumenten ein CONS-Paar, "CONSt" dieses an die gegebene A-Liste vorne an und übergibt die resultierende A-Liste als Wert.

Seiteneffekt: Keiner

Beispiel:
```
-->(ACONS 'MOTHER 'SUE '((FATHER . FRANK)))
((MOTHER . SUE) (FATHER . FRANK))
```

Eine Funktion OUR-ACONS kann definiert werden durch:

```
(DEFUN OUR-ACONS (K D A)
       (CONS (CONS K D) A))
```

Das aus den Werten von K und D gebildete CONS-Paar wird an das von A dargestellte Objekt "angeCONSt". Ob A eine A-Liste darstellt, wird in OUR-ACONS allerdings nicht geprüft.

Die Funktion **PAIRLIS** bildet aus zwei gleichlangen Listen für Schlüssel und Daten eine Assoziationsliste und fügt diese an eine optional gegebene A-Liste an. Sie hat das
Aufrufmuster:

(PAIRLIS <-Schlüsselliste-> <-Datenliste-> <-A-Liste->)

Die ersten beiden Argumente haben als Werte Listen gleicher Länge. Das dritte Argument ist optional und hat als Wert eine A-Liste; der Defaultwert ist NIL.

Auswertung:
Die Elemente der Schlüsselliste werden mit entsprechenden Elementen der Datenliste zu Assoziationspaaren zusammengefaßt. Beide Listen müssen gleiche Länge haben, sonst erfolgt eine

Fehlermeldung. Wenn das optionale Argument vorhanden ist, wird die dadurch gegebene A-Liste vorne um die neuen Assoziationspaare erweitert. Diese neue Liste wird als **Wert** übergeben. Fehlt das optionale Argument, so wird stattdessen der Defaultwert NIL zugrundegelegt. In diesem Fall werden also die Assoziationspaare zu einer A-Liste zusammengefaßt, die als **Wert** übergeben wird. In welcher Reihenfolge die neu angefügten Assoziationspaare vorne in der resultierenden A-Liste erscheinen, ist nicht festgelegt.

Seiteneffekt: Keiner

Beispiel:

```
-->(PAIRLIS '(ONE TWO) '(1 2) '((THREE . 3) (FOUR . 4)))
((ONE . 1) (TWO . 2) (THREE . 3) (FOUR . 4))
oder
((TWO . 2) (ONE . 1) (THREE . 3) (FOUR . 4))
```

A-Listen erfüllen die Aufgabe einer Datenbank. Zur Entnahme von Informationen aus dieser Datenbank dienen die Funktionen ASSOC und RASSOC mit dem
Aufrufmuster:

(ASSOC <Schlüssel> <-A-Liste->)

(RASSOC <Datum> <-A-Liste->)

Das erste Argument ist jeweils ein beliebiger Ausdruck, dessen Wert ein Schlüssel bzw. ein Datum ist; das zweite Argument hat als Wert eine A-Liste.

Auswertung:
Die A-Liste wird von vorne (also von links) her durchsucht; ASSOC übergibt als **Wert** das erste Assoziationspaar, dessen Schlüssel mit dem Wert des ersten Arguments übereinstimmt.

RASSOC übergibt als **Wert** das erste Assoziationspaar, dessen Datum mit dem Wert des ersten Arguments übereinstimmt. Wird ein Assoziationspaar der gesuchten Eigenschaft nicht gefunden, so wird jeweils der Wert NIL übergeben. Der Test auf Übereinstimmung wird jeweils mit EQL durchgeführt, falls nichts anderes festgelegt ist.

Seiteneffekt bei ASSOC und RASSOC: Keiner

Beispiele:

```
-->(SETQ VALUES '((Y . 20) (X . 25) (X . 4) (Z . 5)))
((Y . 20) (X . 25) (X . 4) (Z . Y))

-->(ASSOC 'X VALUES)
(X . 25)

-->(RASSOC '4 VALUES)
(X . 4)
```

Zur Übung definieren wir auch selbst eine Funktion OUR-ASSOC :

```
(DEFUN OUR-ASSOC (KEY A-LIST)
   (COND    ((NULL A-LIST) 'NIL)
            ((EQL KEY    (CAR (CAR A-LIST)))
                         (CAR A-LIST))
            (T (OUR-ASSOC KEY (CDR A-LIST)))))
```

OUR-ASSOC ist eine rekursive Funktion. Falls die gegebene A-Liste leer ist, wird NIL übergeben. Andernfalls wird das erste Assoziationspaar bestimmt und sein Schlüssel, d.h. seine CAR-Komponente mit dem gegebenen Schlüssel KEY verglichen. Liegt Übereinstimmung vor, so wird das erste Assoziationspaar als Wert übergeben. Andernfalls wird OUR-ASSOC auf die nach Entfernen des ersten Assoziationspaares verbleibende A-Liste angewandt.

11.4 Kopieren von Listen

Bei Anwendungen ist es häufig nötig, Listen und die ihnen zugrundeliegenden Speicherstrukturen kopieren zu können. Dazu stellt jedes LISP-System verschiedene Kopierfunktionen bereit. Sie unterscheiden sich vor allem darin, wie weit sie in die Listen- bzw. Baumstruktur hinabsteigen. Bei einigen Kopierfunktionen können auch zusätzliche Substitutionen ausgeführt werden.

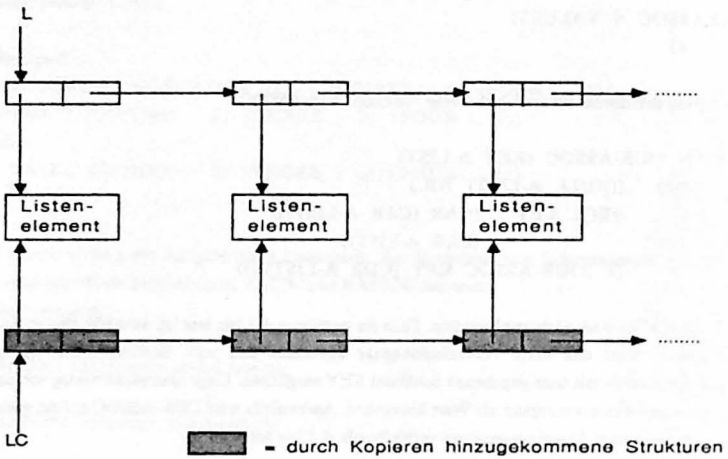
Eine der einfachsten Funktionen zum Kopieren einer Liste ist COPY-LIST mit dem **Aufrufmuster:**

```
(COPY-LIST <-Liste->)
```

Der Wert des Argumentes muß also eine Liste sein.

Auswertung:

COPY-LIST liefert als Wert eine Liste, die mit der gegebenen Liste bezüglich EQUAL übereinstimmt, nicht jedoch bezüglich EQ. Dabei wird nur das oberste Niveau der Listenstruktur kopiert; COPY-LIST kopiert also mit jeder CONS-Zelle auch ihre CDR-Nachfolge, nicht aber ihre CAR-Nachfolge.



Demnach bewirkt

(SETQ LC (COPY-LIST L))

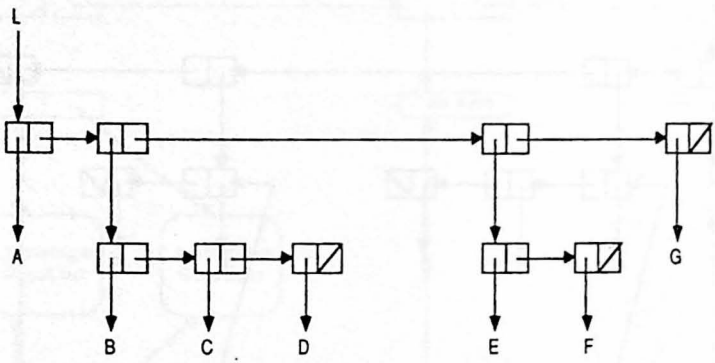
die Anlage der zusätzlichen von LC ausgehenden CONS-Zellen. Zur Kopie des obersten Listenniveaus kommt es durch Anlage von Kopien für die oberste CONS-Zelle und jeder CONS-Zelle, die unmittelbare CDR-Nachfolgerin einer kopierten CONS-Zelle ist.

Beispiel:

Bei Auswertung von

```
-->(SETQ L (A (B C D) (E F) G))
(A (B C D) (E F) G)
```

wird als Seiteneffekt eine interne Struktur für L wie folgt eingerichtet:



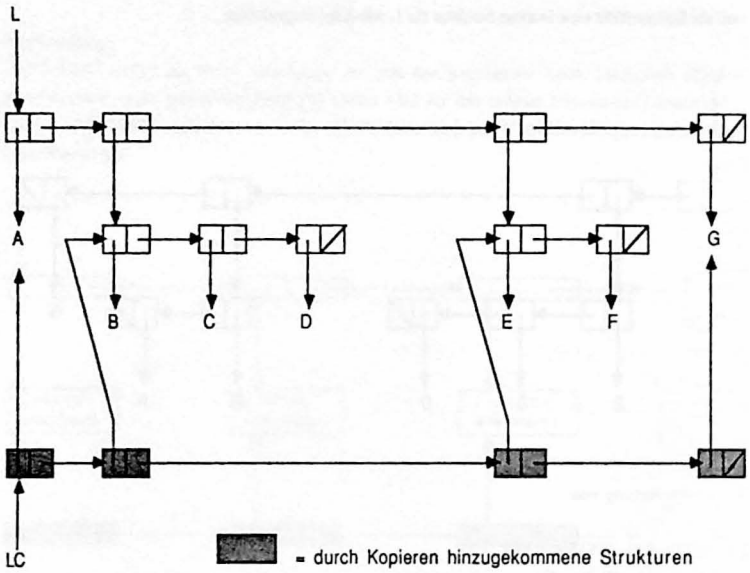
Durch Auswertung von

```
-->(SETQ LC (COPY-LIST L))
(A (B C D) (E F) G)
```

werden zur internen Struktur von L noch Kopien der CONS-Zellen für das oberste Niveau angelegt und die Bindung von LC an die Kopie hergestellt, so daß die nachfolgend dargestellte Gesamtstruktur entsteht.

Illustrativ sind folgende Vergleiche:

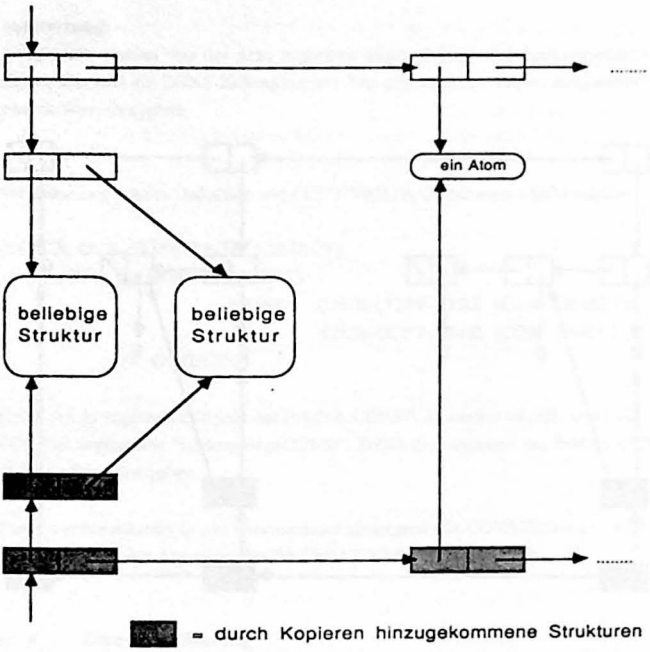
```
-->(EQ L LC)
NIL
-->(EQ (CADR L) (CADR LC))
T
-->(EQ (CADDR L) (CADDR LC))
T
```



Zum Kopieren von A-Listen dient die Funktion COPY-ALIST mit dem
Aufrufmuster:

(COPY-ALIST <-Liste->)

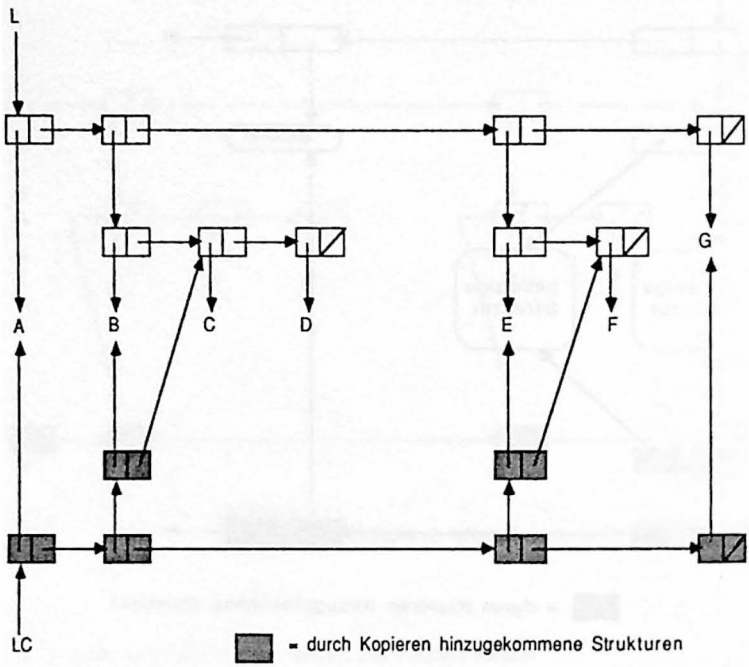
Auswertung:
Zunächst werden die CONS-Zellen für das oberste Niveau kopiert wie bei COPY-LIST. Zusätzlich wird für jede dieser CONS-Zellen die unmittelbare CAR-Nachfolge geprüft: Ist sie eine CONS-Zelle, so wird sie kopiert, deren CAR- und CDR-Nachfolger aber nicht mehr. Auf diese Weise wird die Struktur der Assoziationsliste kopiert, aber nicht die Struktur für ihre Schlüssel und Daten.



Beispiel:
Nach Auswertung von

```
-->(SETQ L '(A (B C D) (E F) G))
(A (B C D) (E F) G)

-->(SETQ LC (COPY-ALIST L))
(A (B C D) (E F) G)
```

liegt nachfolgende interne Struktur vor:



Die oben beschriebenen Kopierfunktionen kopieren nur gewisse Teile der einer Liste zugrundeliegenden Baumstruktur. Eine Funktion, welche die volle Baumstruktur kopiert, ist COPY-TREE mit dem **Aufrufmuster:**

(COPY-TREE <Ausdruck>)

Als Argument ist hier ein beliebiger Ausdruck zugelassen.

Auswertung:
Im Speicher werden von der dem Argument zugrundeliegenden Speicherstruktur alle inneren Knoten (das sind die CONS-Zellen) kopiert. Die vom kopierten Objekt dargestellte Listenstruktur wird als Wert übergeben.

Wir geben eine präzise Definition von COPY-TREE in Gestalt einer LISP-Funktion:

```
(DEFUN OUR-COPY-TREE (OBJECT)
   (COND    ((CONSP    OBJECT)
                       (CONS    (OUR-COPY-TREE (CAR OBJECT))
                                (OUR-COPY-TREE (CDR OBJECT))))
            (T OBJECT)))
```

Erfüllt das zu kopierende Objekt das Prädikat CONSP, so werden rekursiv sein CAR-Teil und sein CDR-Teil kopiert und "zusammengeCONSt". Erfüllt das Argument das Prädikat CONSP nicht, so wird sein Wert übergeben.

Damit werden rekursiv in der Baumstruktur absteigend alle CONS-Zellen kopiert; die Rekursion bricht ab, sobald das Argument das Prädikat CONSP nicht mehr erfüllt.

11.5 Zusammenfassung

Die grundlegende Datenstruktur in LISP, nämlich Listen, werden intern durch binäre Bäume dargestellt, deren Verzweigungen mit CONS-Zellen realisiert werden.

Listen werden auch zur Darstellung von Mengen herangezogen. Verschiedene Mengenoperationen werden mit entsprechenden Listenoperationen simuliert.
Assoziationslisten dienen der Darstellung und Manipulation von dynamischen Paarbeziehungen zwischen Symbolen (Schlüsseln) und Werten. Neueinträge mit PAIRLIS oder ACONS erfolgen stets am vorderen Ende. Die Suche nach Paaren mit bestimmten Eigenschaften wird mit Hilfe von ASSOC oder RASSOC durchgeführt; sie beginnt ebenfalls vorne.
Entfernen von Einträgen ist vorne durch Anwenden von CDR leicht möglich.
Grundsätzlich können aber alle Listenfunktionen zur Manipulation eingesetzt werden.

Verschiedene Funktionen zum Kopieren von Listen unterscheiden sich vor allem darin, welche Teile der internen Struktur dabei neu angelegt werden:

COPY-TREE kopiert den ganzen Baum,
COPY-LIST kopiert nur die das oberste Listenniveau definierenden CONS-Zellen.
Andere Kopierfunktionen sind auf speziellere Bedürfnisse zugeschnitten.

KAPITEL 12:

Destruktive Funktionen

Wir haben bisher eine Vielzahl von Funktionen kennengelernt. Unsere besondere Aufmerksamkeit galt stets den möglichen Seiteneffekten, da Seiteneffekte häufig unliebsame Überraschungen verursachen, aber andererseits auch die einzige Möglichkeit darstellen, um Speicherstrukturen zu verändern.

Soweit bisher Seiteneffekte auftraten, waren sie mit der Anwendung der Funktionen SETQ, DEFVAR und DEFUN sowie LOAD, OPEN, CLOSE, READ und PRINT (einschließlich seiner Variationen) verknüpft. Diese betreffen den Zusammenhang zwischen Variablen und Werten bzw. Funktionen, sowie die Ein- und Ausgabe.
LISP kennt noch eine Reihe weiterer Systemfunktionen, die aufgrund ihrer Seiteneffekte vorhandene Speicherstrukturen in erheblichem Maße verändern können. Man spricht dabei häufig auch von destruktiven Funktionen, obwohl eine klare Abgrenzung der destruktiven Funktionen gegenüber anderen Funktionen mit Seiteneffekten eigentlich nicht existiert. Die Veränderung von den Argumenten zugeordneten Speicherstrukturen kann ein erwünschter Effekt sein, wie z.B. bei SETQ oder einfach nur aus Effizienzgründen in Kauf genommen werden.

Wir wissen, daß Funktionen mit Seiteneffekten nur unter großer Vorsicht eingesetzt werden sollten, da die Auswirkungen schwer überschaubar werden können, wenn verschiedene Listenstrukturen Teile gemeinsam haben.

12.1 RPLACA und RPLACD ändern CAR und CDR

Die Bausteine, aus denen ein Ausdruck beim Aufbau mit CONS aufgebaut wird, können wir mit CAR und CDR zurückgewinnen. Wir können sie aber auch ändern. Mit Hilfe der Systemfunktionen RPLACA (gesprochen Replace CAR) und RPLACD (gesprochen Replace CDR) kann die CAR-Komponente bzw. CDR-Komponente eines Ausdrucks durch einen beliebigen Ausdruck ersetzt werden.

Aufrufmuster:
(RPLACA <-CONS-Paar-> <Ausdruck>)

(RPLACD <-CONS-Paar-> <Ausdruck>)

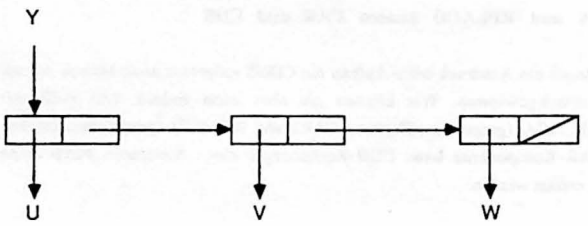
Das erste Argument muß ein Ausdruck sein, der das Prädikat CONSP erfüllt, d.h. sein Wert muß ein CONS-Paar sein.

Wert der RPLACA-Form
ist derjenige Ausdruck, den man erhält, wenn man in dem durch das erste Argument gegebenen CONS-Paar den CAR-Teil ersetzt durch den Wert des zweiten Arguments.

Wert der RPLACD-Form
ist derjenige Ausdruck, den man erhält, wenn man in dem durch das erste Argument gegebenen CONS-Paar den CDR-Teil erseetzt durch den Wert des zweiten Arguments.

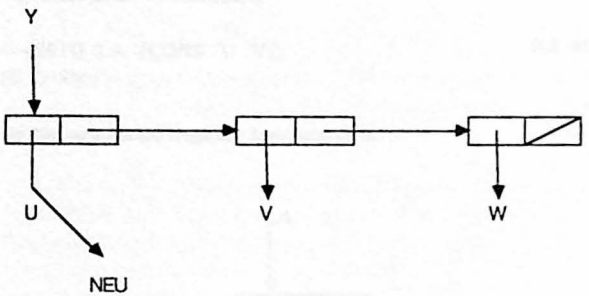
Seiteneffekt:
Der Wert des ersten Arguments wird verändert:
RPLACA ersetzt den CAR-Teil des ersten Arguments durch den Wert des zweiten Arguments.
RPLACD ersetzt den CDR-Teil des ersten Arguments durch den Wert des zweiten Arguments.

Beispiel:
Bei Auswertung von
-->(SETQ Y '(U V W))
(U V W)
wird folgende Struktur aufgebaut:

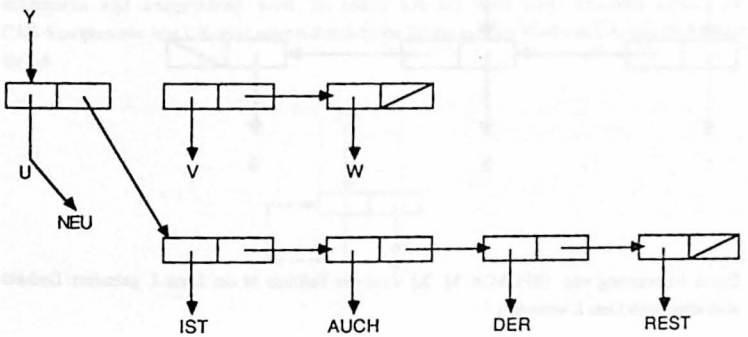


Auswertung von
-->(RPLACA Y 'NEU)
(NEU V W)

bewirkt, daß in der internen Darstellung der Zeiger auf U durch einen Zeiger auf das Symbol NEU ersetzt wird.



Die anschließende **Auswertung** von
-->(RPLACD Y '(IST AUCH DER REST))
(NEU IST AUCH DER REST)
bewirkt, daß in der internen Darstellung der Zeiger auf den bisherigen Listenrest ersetzt wird durch einen Zeiger auf die Darstellung der Liste
(IST AUCH DER REST).

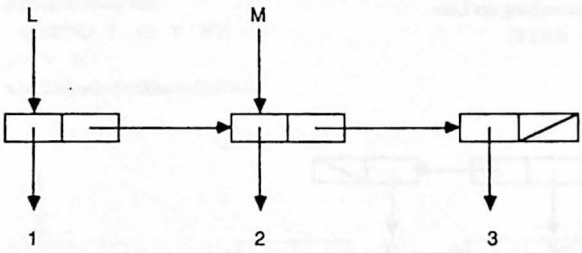


Die mit RPLACA und RPLACD verbundenen Seiteneffekte haben oft unerwünschte Auswirkungen auf andere Variable und Programmteile, die man schwer überblicken kann.

Beispiel:

```
-->(SETQ  L  '(1  2  3))
(1  2  3)

-->(SETQ  M  (CDR  L))
(2  3)

-->(RPLACA  M  'X)
(X  3)

-->M
(X  3)

-->L
(1  X  3)
```

Nach Auswertung der beiden SETQ-Formen liegt folgende interne Struktur vor:



Durch Auswertung von (RPLACA M 'X) wird die Teilliste M der Liste L geändert. Dadurch wird aber auch Liste L verändert.

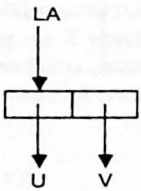
12.2 Zirkuläre Listen

Destruktive Funktionen eröffnen überraschende Möglichkeiten zur Erzeugung "zirkulärer" Listenstrukturen.

Wir betrachten die Wertzuweisung

```
-->(SETQ LA (CONS 'U 'V))
(U . V)
```

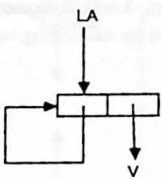
Sie führt zum Aufbau folgender Speicherstruktur.



Wenn jetzt

(RPLACA LA LA)

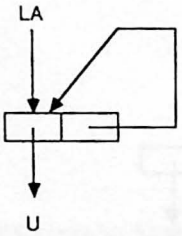
aufgerufen und ausgewertet wird, so erhält LA als Wert eine "zirkuläre Liste"; die CAR-Komponente von LA wird ersetzt durch einen Zeiger auf den Wert von LA, d.h. die Struktur für LA.



Die Ausgabe dieser zirkulären Liste als Wert der RPLACA-Form endet mit einem Disaster: Zunächst wird eine (unendliche !) Folge von linken Klammern erzeugt; abhängig vom jeweiligen LISP-System erfolgt ein Überlauf, eine Meldung "Rechenzeit zu Ende", oder einfach ein Systemzusammenbruch.

Auch die Auswertung von
(RPLACD LA LA)

führt auf eine "zirkuläre Struktur"; jetzt wird die CDR-Komponente von LA durch einen Zeiger auf den Wert von LA ersetzt; die Ausgabe bereitet ähnliche Probleme wie oben.



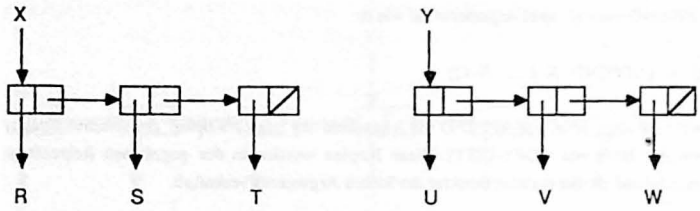
Zirkuläre Strukturen haben u.a. Bedeutung bei der sogenannten Wissensverarbeitung, einem sehr aktuell gewordenen Teilgebiet der Künstlichen Intelligenz.

12.3 Wie arbeitet eigentlich APPEND ?

Die Funktion APPEND fügt - wie wir wissen - Listen zusammen. Wir möchten jetzt erfahren, wie das intern realisiert wird. Dazu betrachten wir die Auswertung von

```
-->(SETQ X '(R S T))
(R S T)
-->(SETQ Y '(U V W))
(U V W)
-->(SETQ A (APPEND X Y))
(R S T U V W)
```

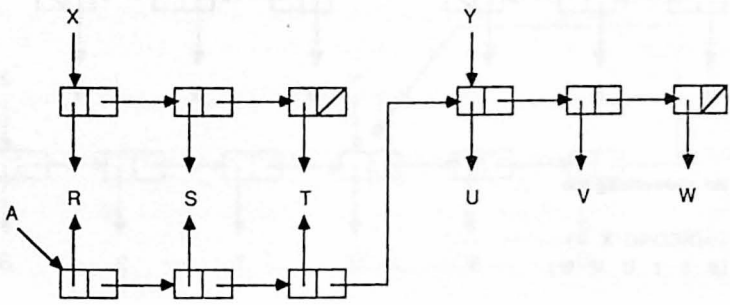
und ihren Effekt auf die interne Struktur:
Man könnte annehmen, daß die Listenstrukturen für X und Y nämlich



bei Auswertung von APPEND einfach verbunden werden, indem das abschließende NIL der Darstellung für X durch einen Zeiger auf die Darstellung für Y ersetzt wird.
Dies würde jedoch den Wert von X verändern. X bekäme den Wert (R S T U V W).
Eine Funktion mit diesem Seiteneffekt ist gefährlich !
Die Funktion APPEND ist gutmütiger und bewirkt folgendes:

Im Zuge der Auswertung von
(SETQ A (APPEND X Y))
wird

- für das Argument X von APPEND eine Kopie der internen Struktur angelegt,
- eine Verbindung zur internen Struktur für das zweite Argument Y hergestellt,
- die Zuordnung der Variablen A an die so aufgebaute Struktur eingerichtet.



Bei Auswertung einer weiteren Wertzuweisung
(SETQ B (APPEND X Y))
wird der entsprechende Aufbau mit einer neuen Kopie von X durchgeführt.

Wenn APPEND mehr als zwei Argumente hat wie in

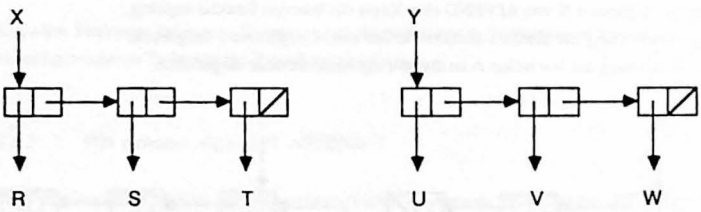
(SETQ A (APPEND X-1 ... X-n))

werden für die Argumente von APPEND mit Ausnahme des letzten Kopien der internen Struktur angelegt (mit Hilfe von COPY-LIST). Diese Kopien werden in der gegebenen Reihenfolge untereinander und mit der internen Struktur des letzten Arguments verknüpft.

12.4 NCONC, die destruktive Variante von APPEND

Es gibt eine Funktion, nämlich NCONC, die Listen gerade so zusammenfügt, wie wir dies bei APPEND als problematisch bezeichnet hatten.

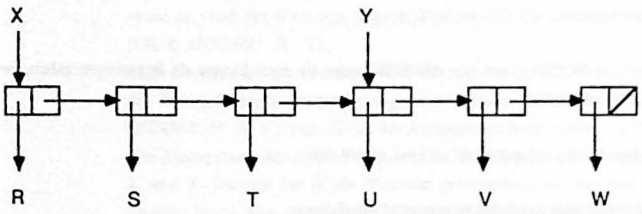
Wir nehmen an, die Symbole X und Y haben als Werte die Listen (R S T) und (U V W), d.h. es liegt folgende interne Struktur vor:



Bei Auswertung von

-->(NCONC X Y)
(R S T U V W)

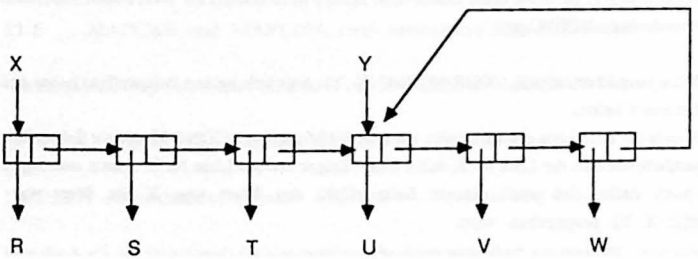
wird jetzt als Seiteneffekt das abschließende NIL in der Interndarstellung von X ersetzt durch einen Zeiger auf die Interndarstellung von Y. Damit entsteht die folgende Struktur mit einem veränderten Wert für X:



Auswertung von X und Y führt jetzt zu folgenden Ergebnissen:

```
-->X
(R  S  T  U  V  W)
-->Y
(U  V  W)
```

Würde man jetzt nochmals
(NCONC X Y)
auswerten, so entstünde eine zirkuläre Listenstruktur der folgenden Gestalt:



In einer externen Darstellung, etwa der Art

(R S T U V W U V W U V W ....)

müßte U V W für immer wiederholt werden. Auch dies kann durch die Standard-Ausgabe nicht verarbeitet werden.

Die Systemfunktion NCONC kann wie APPEND mehr als zwei Listen als Argumente haben; sie hat folgendes **Aufrufmuster:**

(NCONC <-Liste-1-> <-Liste-2-> ... <-Liste-n->)

Die Werte aller $n \geq 0$ Argumente sind also Listen.

Wert:
Diejenige Liste, die durch Zusammenfügen (engl. Concatenation) der durch die Argumente gegebenen Listen zu einer einzigen Liste entsteht.

Seiteneffekt:
Der Wert aller Argumente, mit Ausnahme des letzten, wird verändert:
In der Interndarstellung der Argumente wird in jeder Liste, mit Ausnahme der zum letzten Argument gehörigen Liste, im jeweils letzten CONS-Paar das abschließende NIL ersetzt durch einen Zeiger auf die Interndarstellung der nächsten Liste.

Wir definieren im folgenden eine Funktion OUR-NCONC, die unter Einschränkung auf 2 Argumente X und Y, die beide echte Listen (also keine Punkt-Listen) als Wert haben, äquivalent zur Systemfunktion NCONC ist.

Falls X die leere Liste ist, soll (OUR-NCONC X Y) natürlich keinen Seiteneffekt haben und den Wert von Y liefern.
Falls X nicht die leere Liste darstellt, rufen wir eine Hilfsfunktion NCONC-H, die als Seiteneffekt das abschließende NIL der Liste für X durch einen Zeiger auf die Liste für Y ersetzt und sorgen dann noch dafür, daß nach diesem Seiteneffekt der Wert von X als Wert von (NCONC X Y) ausgegeben wird.

```
(DEFUN OUR-NCONC (X Y)
   (COND    ((NULL X) Y)
            (T (NCONC-H X Y) X)))
```

In der ersten Klausel der COND-Form wird getestet, ob X die leere Liste darstellt, wenn ja, wird der Wert von Y auch Wert der COND-Form und damit Wert von (OUR-NCONC X Y).
Falls X nicht leer ist, wird die zweite Klausel der COND-Form geprüft und wegen des Testes T auch vollständig abgearbeitet. Dazu werden noch die beiden Formen (NCONC-H X Y) und X in der Reihenfolge ihrer Niederschrift ausgewertet. Die Auswertung der ersten Form bewirkt die Koppelung der Listenstrukturen von X und Y. Danach hat X als Wert die gekoppelte Listenstruktur. Der Wert der zweiten Form, also der neue (!) Wert von X gibt dann den Wert der COND-Form und damit den Wert von (OUR-NCONC X Y).

Jetzt zur Funktion NCONC-H; sie ist definiert durch

```
(DEFUN NCONC-H (X Y)
   (COND    ((NULL (CDR X)) (RPLACD X Y))
            (T (NCONC-H (CDR X) Y))))
```

Zur Auswertung von (NCONC-H X Y) wird zuerst das abschließende NIL in der Liste X gesucht. Dazu wird solange X durch (CDR X) ersetzt, bis der Test (NULL (CDR X)) erfüllt ist. Dann wird der letzte CDR-Teil von X, also das abschließende NIL, durch einen Zeiger auf die Liste für Y ersetzt.

12.5 MAPCAN und MAPCON, zwei destruktive Map-Funktionen

Wir kennen die Map-Funktionen MAPCAR und MAPLIST. Bei Aufrufen der Form

(MAPCAR <Funktionales Objekt> <-Liste-1-> ... <-Liste-n->)
und
(MAPLIST <Funktionales Objekt> <-Liste-1-> ... <-Liste-n->)

wird die durch das funktionale Objekt gegebene Funktion wiederholt ausgewertet:

Bei MAPCAR zuerst für die ersten Elemente (die CAR´s) der n-Listen, dann ihre zweiten Elemente, ihre dritten Elemente usw.
Bei MAPLIST zuerst für die n Listen selbst, dann ihre CDR´s, ihre CDDR´s usw.
Die Resultate dieser sukkzessiven Funktionsanwendungen werden jeweils mit der Systemfunktion LIST zu einer Liste zusammengefaßt, die als Wert der MAPCAR- bzw. MAPLIST-Form übergeben wird.

MAPCAN und MAPCON sind destruktive Varianten von MAPCAR und MAPLIST und unterscheiden sich von diesen allein dadurch, daß die Resultate der sukzessiven Funktionsanwendungen nicht mit der Funktion LIST, sondern mit der destruktiven Funktion NCONC zusammengefaßt werden. Dabei gilt folgende Äquivalenz:

(MAPCAN <Funktionales Objekt> <-Liste-1-> ... <-Liste-n->)
ist äquivalent zu
(APPLY #'NCONC (MAPCAR <Funktionales Objekt> <-Liste-1-> ... <-Liste-n->))

(MAPCAN <Funktionales Objekt> <-Liste-1-> ... <-Liste-n->)
ist äquivalent zu
(APPLY #'NCONC (MAPLIST <Funktionales Objekt> <-Liste-1-> ... <-Liste-n->))

Eine wichtige Anwendung von MAPCAN und MAPCON ist die "Filterung" von Listen: Nur Elemente, die einer bestimmten Bedingung genügen, können den Filter passieren.

Beispiele:
Das **erste Beispiel** zeigt eine Anwendung von MAPCAN; dabei werden aus einer Liste diejenigen Elemente, die Zahlen sind, herausgegriffen und in einer neuen Liste zusammengefaßt.

```
-->(MAPCAN #'(LAMBDA (X)
                (COND ((NUMBERP X) (LIST X)) (T NIL)))
             '(a 2 b c 3 4 d 5))
(2 3 4 5)
```

Der Lambda-Ausdruck liefert den Wert NIL, wenn sein Argument keine Zahl ist. Dieses NIL wird beim Zusammenfassen der Resultate mit NCONC absorbiert und verschwindet.
Ist das Argument aber eine Zahl, so wird eine nur aus dieser Zahl bestehende Liste gebildet. Diese wird übergeben und mit NCONC in die Resultatliste eingebracht.

Das zweite Beispiel zeigt eine Filterung unter komplexeren Bedingungen mit drei Argumentlisten.

Ein Element aus der ersten Argumentliste wird dann ausgewählt, wenn die in zwei weiteren Argumentlisten an entsprechender Stelle stehenden Elemente eine bestimmte Bedingung erfüllen. Die ausgewählten Elemente werden in einer Liste zusammengefaßt.

```
-->(MAPCAN   #'(LAMBDA (X Y Z)
                 (COND ((< Y Z) (LIST X)) (T NIL)))
             '(4711 KOELN AACHEN WIEN)
             '(41  16  34  38)
             '(43  19  27  36))
(4711 KOELN)
```

Das nächste Beispiel illustriert MAPCON.
Wir definieren hier eine Funktion, die aus einer Liste von Symbolen alle Duplikate entfernt.

```
-->(DEFUN DUPLIKATE (LISTE)
     (MAPCON #'(LAMBDA (LISTE-1)
                 (COND ((MEMBER (CAR LISTE-1) (CDR LISTE-1)) NIL)
                       (T (LIST (CAR LISTE-1))))) LISTE))
DUPLIKATE

-->(DUPLIKATE '(A B C C A H F B H))
(C A F B H)
```

Die durch den anonymen Lambda-Ausdruck definierte Funktion prüft zunächst, ob das Element (CAR LISTE-1) in der Liste (CDR LISTE-1) enthalten ist. Wenn ja, wird NIL übergeben; wenn nein, wird eine nur aus dem Element (CAR LISTE-1) bestehende Liste gebildet und als Wert übergeben.
MAPCON bewirkt, daß dies der Reihe nach für LISTE, dann für (CDR LISTE), dann für (CDDR LISTE) usw. durchgeführt wird und daß die resultierenden Werte mit NCONC zusammengefaßt werden.

12.6 NREVERSE zur Umkehrung von Listen

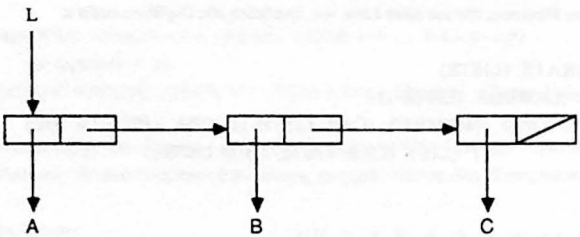
Wir haben die Systemfunktion REVERSE kennengelernt.

Ein Aufruf der Gestalt

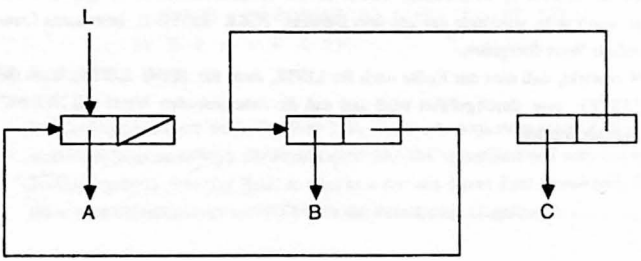
(REVERSE L)

für ein Symbol L, dessen Wert eine Liste ist, liefert als Wert die Umkehrung dieser Liste, ohne dabei die Speicherstruktur für L zu verändern. Dabei arbeitet REVERSE i.a. sehr ineffizient. Die Umkehrung einer Liste kann wesentlich effizienter ermittelt werden, wenn man in Kauf nimmt, daß die Speicherstruktur für die ursprüngliche Liste zerstört wird.

Zum Beispiel kann aus einer Speicherstruktur der Gestalt



durch Umsetzung der Zeiger eine Struktur gewonnen werden, die gerade die Umkehrung des ursprünglichen Wertes von L darstellt.



In dieser Weise arbeitet die Funktion NREVERSE.

Ein Aufruf der Gestalt
(NREVERSE <-Liste->)

mit einem Argument, dessen Wert eine Liste ist, liefert als Wert die Umkehrung dieser Liste.
Als Seiteneffekt kann dabei der Wert von L verändert werden. Die Art der Veränderung wird in COMMON LISP nicht festgelegt. Damit ist folgendes möglich:

```
-->(SETQ L '(A B C))
(A B C)
-->(NREVERSE L)
(C B A)
->L
(A)
```

Da nicht festgelegt ist, welchen Wert L nach Auswertung von (NREVERSE L) haben muß, ist es häufig zweckmäßig, NREVERSE in einer Kombination wie

```
(SETQ L (NREVERSE L))
```

aufzurufen.

12.7 Zusammenfassung

Destruktive Funktionen verändern die ihren Argumenten zugeordneten Speicherstrukturen. Sie ermöglichen "chirurgische Eingriffe" in bestehende Strukturen und beinhalten auch entsprechende Risiken.

Mit RPLACA und RPLACD lassen sich die CAR- und CDR-Bestandteile von CONS-Zellen "herausschneiden" und ersetzen. Mit NCONC können Listen miteinander "vernäht" werden. Zusammenfassen der Teilresultate mit NCONC unterscheidet die destruktiven MAP-Funktionen MAPCAN und MAPCON von MAPCAR und MAPLIST.

Funktionen mit Seiteneffekten werden nicht immer nur ihres Seiteneffekts wegen eingesetzt; häufig wird die Zerstörung von Speicherstrukturen nur deshalb in Kauf genommen, weil auf diese Weise Zeit und Speicher gespart werden können. Ein Beispiel ist NREVERSE , eine destruktive Variante von REVERSE.

NREVERSE zerstört eine Liste und bildet eine neue, indem es die Bestandteile der alten in umgekehrter Reihenfolge "vernäht". Und das geht sehr effizient, man könnte sagen (NREVERSE '(KOPF UEBER HALS)).
LISP kennt noch weitere destruktive Funktionen.

KAPITEL 13:

Formatierte Ausgabe

Wir haben in Kapitel 10 die Grundlagen der Ein- und Ausgabe kennengelernt. Wenn umfangreiches Datenmaterial auszugeben ist, hängt die Lesbarkeit der Ausgabe stark von ihrer Gestaltung ab. Mit dieser Gestaltung der Ausgabe, d.h. mit ihrer Formatierung wollen wir uns jetzt beschäftigen: Zuerst lernen wir noch einige Möglichkeiten der Steuerung bei den bereits bekannten Ausgabefunktionen PRINT, PRIN1 und PRINC kennen und gehen dann kurz auf den sogenannten Pretty-Printer ein. Dann behandeln wir FORMAT, eine sehr vielseitige und flexible, aber auch komplizierte Ausgabefunktion. Wir können allerdings hier nur ihre wichtigsten Möglichkeiten aufzeigen. Anschließend lernen wir die von FORMAT abgeleitete Funktion ERROR kennen. Mit ihr kann der Benutzer Möglichkeiten der Fehlerdiagnose in selbstdefinierte Funktionen einprogrammieren.

13.1 Mehr zu PRINT, PRIN1 und PRINC

Wir haben die "PRINT-Funktionen" **PRINT, PRIN1** und **PRINC** kennengelernt. Die jeweils produzierte Ausgabe hängt allerdings noch von den globalen Variablen

PRINT-LEVEL, *PRINT-LENGTH* und *PRINT-BASE* ab:

PRINT-LEVEL

bestimmt für die Ausgabe von Listen die maximal wiedergegebene Schachtelungstiefe. Ist der Wert gleich NIL, so ist die wiedergegebene Schachtelungstiefe nicht beschränkt.
Ist der Wert eine positive ganze Zahl n, so werden bei der Ausgabe von Listen alle Teile bis in die Schachtelungstiefe n ausgedruckt, aber für jedes tiefer eingeschachtelte Teilstück das Zeichen # ausgegeben.

PRINT-LENGTH

bestimmt für die Ausgabe von Listen die maximale Anzahl der wiedergegebenen Listenelemente.
Ist der Wert gleich NIL, so ist die Anzahl der wiedergegebenen Listenelemente nicht beschränkt.

Ist der Wert eine positive ganze Zahl n, so werden von jeder Liste maximal n Listenelemente ausgedruckt; für darüberhinaus noch vorhandene Listenelemente werden nur 3 Punkte ausgegeben.

PRINT-BASE

bestimmt für die Ausgabe von ganzen Zahlen die Basis der Zahldarstellung.

Die Variablen ***PRINT-LEVEL*** und ***PRINT-LENGTH*** werden vom LISP-System grundsätzlich mit NIL - also für ungekürzten Listenausdruck - initialisiert.

PRINT-BASE wird vom LISP-System grundsätzlich mit 10 - also für Dezimaldarstellung - initialisiert.

Beispiele:

```
-->(SETQ  *PRINT-LEVEL*  2)
2

-->'((AD)  ((BC)  A  (BC))  (M  (FG)))
((AD)  (# A #)  (M #))

-->(SETQ *PRINT-LEVEL* NIL)
NIL

-->(SETQ  *PRINT-LENGTH*  3)
3

-->'(A  B C  D E)
(A  B C  ...)

-->'(A  (B  C  (D E  F  G)  H)  (I  J  K  L)  M)
(A  (B  C  (D  E  F ...) ...)  (I  J  K ...) ...)

-->(SETQ *PRINT-LEVEL* 2)
2

-->'(A  (B  C)  (D  E  F  G)  H)  (I  J  K  L)  M)
(A  (B  C  # ...)  (I  J  K ...) ...)
```

```
-->(SETQ *PRINT-BASE* 8)
10                          ; Das zweite Argument, die Zahl "acht" im Dezi-
                            ; malsystem, wird im Oktalsystem, d.h. bei Zugrunde-
                            ; legung der Basis acht durch die Ziffernfolge 10
                            ; dargestellt

-->(+ 7 7)
16                          ; "vierzehn" im Oktalsystem

-->(* 5 13)
101                         ; "fünfundsechzig" im Oktalsystem
```

Wir erwähnen, daß eine globale Variable *READ-BASE* die Zahldarstellung beim Einlesen ganzer Zahlen in ähnlicher Weise kontrolliert wie *PRINT-BASE* beim Drucken.
Dabei gelten folgende Regeln:
Ziffernfolgen, die keinen Dezimalpunkt enthalten, werden als ganze Zahlen interpretiert, allerdings in dem durch den Wert von *READ-BASE* bestimmten Zahlsystem.
Jede Zahl, in der ein Dezimalpunkt auftaucht, wird vom System als Dezimalzahl interpretiert.
READ-BASE wird vom LISP-System grundsätzlich mit 10 initialisiert.

Beispiel:

```
-->(SETQ *PRINT-BASE* 10.)      ; der Dezimalpunkt beim zweiten Argument sichert die
10                              ; Interpretation als Dezimalzahl

-->(SETQ *READ-BASE* 8.)
8

-->(LIST 63 63. 100 100.1)      ; das zweite und vierte Argument enthält einen
(51 63 64 100.1)                ; Dezimalpunkt und wird deshalb als Dezimalzahl
                                ; interpretiert,
                                ; das erste und dritte Argument dagegen als Zahl zur
                                ; Basis 8;
                                ; für die Ausgabe wurde oben Darstellung mit Basis 10
                                ; festgelegt.
```

13.2 "Schönheit überzeugt leichter !"

Alte Chinesische Volksweisheit

Häufig sind LISP-Ausdrücke so umfangreich und kompliziert, daß sie schwer zu verstehen sind, wenn sie mit PRINT, PRIN1 oder PRINC ausgegeben werden.
Deshalb verwendet man sogenannte Pretty-Printer, die LISP-Ausdrücke in besser lesbarer Gestalt ausgeben. In erster Linie geht es dabei um eine Ausgabe, die Klammerstrukturen durch entsprechendes Einrücken aufzeigt. COMMON-LISP kennt dazu die Funktion PPRINT (von Pretty Print) mit dem
Aufrufmuster :

(PPRINT <Form> <Strom>)

Die Argumente haben dieselbe Bedeutung wie bei PRINT; das zweite Argument ist wieder optional mit dem Defaultwert T, der auf die Standard-Ausgabe führt.
Als **Seiteneffekt** wird eine "gefällige" Ausgabe erzeugt. Worin die Schönheit einer Ausgabe besteht, ist in COMMON-LISP nicht genau spezifiziert, sondern dem jeweiligen System-Designer zur Entscheidung überlassen.
Eine PPRINT-Form liefert übrigens **keinen Wert.**

13.3 Die Funktion FORMAT

Für formatierte Ausgabe wurde in COMMON-LISP die Systemfunktion FORMAT eingeführt.

Sie kann aufgerufen werden mit dem
Aufrufmuster:

(FORMAT <Bestimmung> <Steuerstring> <Arg-1> ... <Arg-n>) mit $n \geq 0$

Das Argument Bestimmung bestimmt, wohin die Ausgabe geleitet werden soll.
Das Argument Steuerstring legt zusammen mit den restlichen Argumenten Arg-1 bis Arg-n Gestaltung und Inhalt der Ausgabe fest.
Falls Bestimmung den Wert NIL hat, so wird ein String als Wert des Aufrufs von FORMAT übergeben; in diesem Fall hat FORMAT keinen Seiteneffekt.

Hat Bestimmung einen Wert ungleich NIL, so übergibt FORMAT den Wert NIL, bewirkt aber als Seiteneffekt eine Ausgabe. Der Weg dieser Ausgabe ist durch Bestimmung festgelegt.
Besonders wichtig ist dabei der Fall, daß Bestimmung einen Strom für die Ausgabe festlegt.
Falls Bestimmung den Wert T hat, so erfolgt die Ausgabe über denjenigen Strom, der Wert der globalen Variablen *STANDARD-OUTPUT* ist.

Zur Erinnerung:

Ein String in COMMON LISP ist eine Zeichenfolge, die mit hochstehenden Anführungszeichen eingerahmt ist wie "abcde", "653", "LAMBDA" oder "~%". Ein String ist ein Atom und hat genauso wie Zahlen oder die Atome T und NIL stets sich selbst als Wert, muß also auch nicht gequotet werden.

Wir betrachten drei einfache Beispiele:

```
-->(FORMAT NIL "AUSGABE EINES STRINGS")   ; Bestimmung hat Wert NIL
"AUSGABE EINES STRINGS"                   ; Wert der Form ist ein String

-->(FORMAT T "OHNE ANFUEHRUNGSZEICHEN") ; Bestimmung hat Wert ungleich NIL
OHNE ANFUEHRUNGSZEICHEN                   ; Ausgabe durch Seiteneffekt
NIL                                       ; Wert ist NIL

-->(FORMAT T '"MIT QUOTIERUNG")
MIT QUOTIERUNG
NIL
```

Im ersten Beispiel wird kein Seiteneffekt bewirkt und der Steuerstring als Wert der Form unverändert ausgegeben.
Im zweiten und dritten Beispiel wird als Seiteneffekt der ansonsten unveränderte Inhalt des Strings ausgegeben und danach NIL als Wert der Form ausgegeben.

In allen drei Beispielen wurde der Steuerstring völlig bzw. im wesentlichen unverändert (nur ohne Anführungszeichen am Anfang und Ende) ausgegeben, da er keine Angaben zum Druckbild oder zur Verwendung irgendwelcher Argumente enthielt. Durch Steueranweisungen kann die Ausgabe des Steuerstrings verändert werden. Derartige **Steueranweisungen** werden stets durch das Zeichen ~ (gesprochen Tilde) eingeleitet, dann kommen ein oder mehr Steuerzeichen. Wichtige Steueranweisungen sind ~% und ~<newline> sowie ~S. Das Auftreten von ~% bewirkt,

daß bei der Ausgabe des Steuerstrings an der betreffenden Stelle ein Zeilenvorschub vorgenommen wird. Das Auftreten von ~ unmittelbar vor einem Zeilenvorschub im Steuerstring bewirkt, daß dieser Zeilenvorschub und eventuelle Leerzeichen am Beginn der neuen Zeile ignoriert werden. Das Auftreten von ~S bewirkt, daß bei der Ausgabe des Steuerstrings an der betreffenden Stelle der Wert des nächsten noch nicht benutzten Arguments ausgedruckt wird.

Beispiele:

```
-->(FORMAT T "AUSGABE
                        VERSCHOBEN")
AUSGABE
                        VERSCHOBEN
NIL

-->(FORMAT T "AUSGABE ~
                        NICHT VERSCHOBEN")
AUSGABE NICHT VERSCHOBEN
NIL

-->(FORMAT NIL "AUSGABE ~%MIT ~
                        TSEILENFORSCHUP UNT DIPFÄLER")
"AUSGABE
MIT TSEILENFORSCHUP UNT DIPFÄLER"

-->(SETQ X '(A B C))
(A B C)


-->(FORMAT T "AHA ! ~%LISTE ~S HAT LAENGE ~S"
            X (LENGTH X))       ; Argumente sind X und (LENGTH X)
AHA !
LISTE (A B C) HAT LAENGE 3.
NIL
```

Das folgende Beispiel zeigt, daß auch die Ausgabe eines Strings durch die Steueranweisung ~S veranlaßt werden kann. Wenn wir das Zeichen für Tilde ausgeben möchten, müssen wir die Steueranweisung ~~ verwenden.

```
-->(FORMAT T "~S IST EIN STRING ~%ABER ~~ IST EINE TILDE"
             "ABC")                          ; Argument "ABC"
"ABC" IST EIN STRING
ABER ~ IST EINE TILDE
NIL
```

Der String "ABC" wurde mit Anführungszeichen ausgegeben. Diese werden unterdrückt, wenn man anstatt ~S die Steueranweisung ~A verwendet. ~A veranlaßt, daß der Wert des nächsten noch nicht benutzten Arguments ausgegeben wird, aber unter Zugrundelegung der Druckanweisung PRINC. Diese unterdrückt - wie wir hoffentlich noch wissen - alle Anführungszeichen.

Beispiel:

```
-->(SETQ NOTE 2)
2
-->(FORMAT T "ICH LIEBE LISP ~A ~%"
          (COND    ((= NOTE 1)    "SEHR")
                   ((= NOTE 2)    "WIRKLICH")
                   (T             "AUCH")))
ICH LIEBE LISP WIRKLICH

NIL
```

Die Steueranweisungen **~*** bzw. **~7*** bewirken, daß das nächste Argument bzw. die nächsten 7 Argumente bei der Ausgabe übergangen werden.

Die Steueranweisungen **~:*** bzw. **~4:*** bewirken bei der Kennzeichnung der bereits verarbeiteten Argumente eine Rücksetzung um eine bzw. vier Positionen. Das bedeutet, daß das zuletzt ausgegebene Argument bzw. die vier zuletzt ausgegebenen Argumente nochmals ausgegeben werden.

Die Ausgabe von Zahlen kann sehr flexibel gestaltet werden. Wir betrachten zunächst Steueranweisungen für ganze Zahlen.

~D benötigt ein Argument, das eine ganze Zahl darstellt und veranlaßt deren Ausgabe in Dezimaldarstellung.

~B, ~O, ~X und ~nR mit einer ganzen Zahl n, $2 \leq n \leq 32$, erwarten ebenfalls ein Argument, das eine ganze Zahl darstellt, und veranlassen deren Ausgabe in Binär-, Oktal-, Hexadezimaldarstellung bzw. in Darstellung zur Basis (engl. Radix) n.

~R bzw. ~:R veranlaßt die Ausgabe einer nichtnegativen ganzen Zahl als englische Kardinalzahl (wie four) bzw. als englische Ordinalzahl (wie fourth).

Beispiele:

```
-->(FORMAT NIL "~R in Binärdarstellung: ~*~B" 17) ; Argument 17 zweimal benutzt
"seventeen in Binärdarstellung: 10001"

-->(FORMAT NIL "~R in Hexadezimaldarstellung: ~*~X" 18)
"eighteen in Hexadezimaldarstellung: 12"

-->(FORMAT NIL "~R hat bei Basis 19 die Darstellung: ~*~19R" 24)
"twenty-four hat bei Basis 19 die Darstellung: 15"
```

Wir erwähnen noch folgende Steueranweisungen für Gleitpunktzahlen:

~F	für "floating-point number"
~E	für "floating-pointer number in exponential notation"
~G	für "general floating-point number" (System wählt geeignete Darstellung)
~$	für Dollars floating-point (mit festem Format, besonders geeignet zum Drucken von Geldbeträgen)

Ein Aufruf von FORMAT ist fehlerhaft, wenn die im Anschluß an den Steuerstring angegebenen Argumente nicht für alle Argumentanforderungen aus dem Steuerstring ausreichen. Er ist nicht fehlerhaft, wenn im Aufruf mehr Argumente zur Verfügung stehen, als tatsächlich benötigt werden.

13.4 Weitere Steueranweisungen für FORMAT

Allgemein besteht eine Steueranweisung für FORMAT aus

- dem einleitenden Symbol Tilde, also ~;
- optionalen "Zahlparametern", getrennt durch Komma;

- optionalen "Modifikatoren"; als solche werden
  der Doppelpunkt : und
  das at-Zeichen @
  benutzt
- dem abschließenden Steuerzeichen, das die Art der Steueranweisung beschreibt.

Eine Steueranweisung darf keine Leerzeichen enthalten.

Auch die bisher aufgetretenen Steueranweisungen , stellvertretend seien genannt

~%	zum Übergang auf neue Zeile
~S	zur Ausgabe des Wertes eines S-Ausdrucks, also einer Form
~A	zur Ausgabe des Wertes einer Form
~D	zur Ausgabe von ganzen Zahlen in Dezimaldarstellung
~E	zur Ausgabe von Gleitpunktzahlen in Exponentendarstellung

können mit Parametern und/oder Modifikatoren versehen werden, so bewirkt z.B.

~3%	den 3-maligen Übergang auf eine neue Zeile
~12S	daß unmittelbar vor Ausgabe des Wertes des nächsten Arguments soviele Leerzeichen eingefügt werden, daß diese zusammen mit dem Druckbild für den Wert mindestens 12 Zeichenpositionen belegen,
~22,'*D	daß bei der Ausgabe des anstehenden Arguments, das eine ganze Zahl sein muß, unmittelbar vor der Dezimaldarstellung der Zahl so oft der Stern * eingefügt wird, daß durch diese Sterne und das Druckbild für die Zahl mindestens 22 Zeichenpositionen belegt werden.

Insbesondere die Ausgabe von Gleitpunktzahlen kann mit verschiedenen Parametern und Modifikatoren sehr stark variiert werden.

Für die Ausgabe von Texten gibt es Steueranweisungen, die ein Deklinieren englischer Substantiva bzw. Konjugieren von englischen Verben unterstützen.

Die Steueranweisung

- ~P verbraucht ein Argument; ist dieses nicht EQL zur ganzen Zahl 1, so wird der

Kleinbuchstabe s ausgegeben, ist das Argument EQL zu 1, so wird nichts ausgegeben.

- ~:P fungiert als Abkürzung von ~:*~P (also ~P aber mit Wiederverwendung eines Arguments).
- ~@P verbraucht ein Argument und veranlaßt Ausgabe von y, wenn das Argument den Wert 1 hat, andernfalls Ausgabe von ies.
- ~:@P fungiert als Abkürzung von ~:*~@P (also ~@P mit Wiederverwendung eines Arguments).

Beispiele:

```
-->(FORMAT NIL " ~D tr~:@P/~D win~:P " 7 1)
" 7 tries/1 win "                                   ; 7 Versuche / 1 Gewinn

-->(FORMAT NIL " ~D tr~:@P/~D win~:P " 6 3)
" 7 tries/3 wins "

-->(FORMAT NIL " ~D tr~:@P/~D win~:P " 1 0)
" 1 try/0 wins "
```

Wir erwähnen noch Steueranweisungen für Fallunterscheidungen:

~[<Zfolge-0>~; <Zfolge-1>~; ...~; <Zfolge-n>~]

verbraucht ein Argument, dessen Wert eine der Zahlen 0 bis n sein muß. Hat das Argument den Wert i, so wird die Ausgabe der Zeichenfolge-i veranlaßt.

Hat das Argument einen Wert außerhalb der Zahlenfolge von 0 bis n, so wird weder eine Ausgabe veranlaßt, noch ein Fehler gemeldet. Dagegen berücksichtigt die folgende Steueranweisung auch einen Default-Fall:

~[<Zfolge-0>~; <Zfolge-1>~; ...~; <Zfolge-n>~:; <Default-Zfolge>~]

Hat das Argument als Wert eine ganze Zahl i mit $0 \le i \le n$, so wird, wie oben, die Ausgabe der Zeichenfolge-i veranlaßt. Liegt der Wert des Arguments dagegen außerhalb dieses Bereiches, so wird die Default-Zeichenfolge ausgegeben.

Beispiele:

```
-->(SETQ k 0 m 1 n 2 s 7)
7
```

```
-->(FORMAT NIL  "~D dog~:*~[s are~;  is~;  s are~] here" n)
"2 dogs are here"

-->(FORMAT NIL  "~D dog~:*~[s are~;  is~;  s are~] here" k)
"0 dogs are here"

-->(FORMAT NIL  "~D dog~:*~[s are~;  is~;  s are~] here" m)
"1 dog is here"

-->(FORMAT NIL  "~D dog~:*~[s are~;  is~:;  s are~] here" s)
"7 dogs are here"
```

Für **Fallunterscheidungen** mit if-then-else-Charakter kann man die Steueranweisung

~:[<False-Zfolge>~; <True-Zfolge>~]

verwenden. Sie verbraucht ein Argument und veranlaßt die Ausgabe der False-Zeichenfolge, wenn das Argument den Wert NIL hat, andernfalls die Ausgabe der True-Zeichenfolge.

Beispiele:

```
-->(FORMAT NIL "~D dog~:[s are~; is~] here" s (EQL s 1))
"7 dogs are here"

-->(FORMAT NIL "~D dog~:[s are~; is~] here" m (EQL m 1))
"1 dog is here"
```

Zu obigen Steueranweisungen für Fallunterscheidungen existieren noch weitere Modifikationen, auf die wir aus Platzgründen nicht mehr eingehen.

Zum **Tabellieren** dient die Steueranweisung

~n,mT mit ganzen Zahlen $n \geq 1$, $m \geq 0$.

Sie bewirkt, daß in der jeweils vorliegenden Zeile für den nächsten Eintrag unter Ausgabe von null oder mehr Leerzeichen auf die Zeichenposition n vorgerückt wird, falls diese noch nicht erreicht oder überschritten ist. Ist sie bereits erreicht oder überschritten, so wird unter Ausgabe von einem oder mehr Leerzeichen auf die nächste Zeichenposition $n + km$ mit minimalem k vorgerückt. Dabei sind die Zeichenpositionen einer Zeile von 0 ab durchnumeriert.

Die Parameter n und m sind optional und haben beide als Default-Wert die Zahl 1.
Ist also zum Beispiel beim Auftreten der Steuereingabe ~9T die Zeichenposition mit Nummer 9

bereits überschritten, so rückt der Kursor noch eine Position weiter, d.h. es wird noch ein Leerzeichen eingefügt.

Beispiel:

Wir betrachten die Aufgabe, aufgrund einer Kartei, die Postleitzahlen und zugehörige Städte nennt, eine Tabelle auszugeben, welche zuerst die Städte und dann die Postleitzahlen nennt. Wir verwenden dazu die Tabellieranweisung ~5,2T . Man beachte insbesondere, wie die Postleitzahl im Anschluß an die Stadt ausgegeben wird, wenn der Name der Stadt 5 , 6 oder 7 Buchstaben hat. Anschließend verwenden wir die Steueranweisungen für Fallunterscheidungen und Rücksetzung in den Argumenten, um den Text, "Das war 1 Stadt" bzw. "Das waren i Staedte", wobei i eine der Zahlen 0,2,3, ... ist, jeweils korrekt ausgeben zu können.

```
-->(SETQ LISTE1 '( (6700  MAINZ))
                  (1000  BERLIN)
                  (2000  HAMBURG)
                  (7570  BADEN-BADEN)
                  (6750  KAISERSLAUTERN))
( (6500 MAINZ) (1000 BERLIN)
  (2000 HAMBURG) (7570 BADEN-BADEN) (6750 KAISERSLAUTERN))

-->(SETQ LISTE2 '())
NIL

-->(SETQ LISTE3 '((2000 HAMBURG)))
((2000 HAMBURG))

-->(DEFUN AUSGABE (LISTE)
      DO ((LI LISTE (CDR LI))
          ((NULL LI) (FORMAT T  "~%Das ~:[war~;waren~] ~D ~2:* ~
                                ~:[Stadt~;Staedte~]."
                                (NOT (= (LENGTH LISTE) 1))
                                (LENGTH LISTE)))
          (FORMAT T "~%~S~5,2T~S" (CADAR LI) (CAAR LI))))
AUSGABE
```

```
->(AUSGABE LISTE1)

MAINZ6500
BERLIN 1000
HAMBURG  2000
BADEN-BADEN  7570
KAISERSLAUTERN 6750

Das waren 5 Staedte
NIL

->(AUSGABE LISTE2)

Das  waren  0  Staedte
NIL

->(AUSGABE LISTE3)

HAMBURG  2000
Das  war  1  Stadt
NIL
```

13.5 Fehlermeldungen

Mit Hilfe der ERROR-Funktion kann die Meldung von Fehlern vom Benutzer individuell programmiert werden.
Wir betrachten als Beispiel die Funktion OUR-REVERSE mit der Definition

```
(DEFUN OUR-REVERSE (LST)
  (DO    ((REM LST (CDR REM))
          (AUX 'NIL (CONS (CAR REM) AUX)))
         ((NULL REM) AUX)
                                  ; Rumpf der DO-Form ist leer
                                  ))
```

OUR-REVERSE erwartet als Argument eine Liste; für ein Argument, das keine Liste darstellt, kann es selbst kein korrektes Resultat übergeben. Im allgemeinen übernimmt OUR-REVERSE als Wert für LST das Ergebnis der Auswertung einer anderen Funktion und übergibt seinerseits sein Resultat an eine andere Funktion. Dabei macht sich die Existenz eines Fehlers u.U. erst sehr viel später bemerkbar, was ein Erkennen der Ursachen erschwert. Deshalb wäre es sinnvoll, in Funktionen wie OUR-REVERSE, Abfragen auf Fehler einzubauen und im Fehlerfall entsprechende Informationen auszugeben.
Angenommen, der an LST übergebene Argumentwert ist keine korrekte Liste, und dies wird nicht durch iterierte Anwendung von CDR erkannt. Dann ist das Argument ein CONS-Paar, das keine Liste darstellt. Dann wird es aber durch iterierte Anwendung von CDR nicht zur leeren Liste, sondern zu einem Atom abgearbeitet. In diesem Fall sollte eine Fehlermeldung ausgegeben werden und die Bearbeitung abgebrochen werden.

Dies wird erreicht durch Aufnahme von

```
(COND    ((ATOM REM)
                  (ERROR  "Argument von  OUR-REVERSE ~
                          keine korrekte Liste:  ~S"  LST)))
```

in den Rumpf der DO-Form.

Die Definition von OUR-REVERSE lautet dann

```
-->(DEFUN OUR-REVERSE (LST)
    (DO  ((REM LST (CDR REM))
          (AUX  'NIL (CONS (CAR REM) AUX)))
         ((NULL REM) AUX)
                          ; Rumpf der DO-Form war leer
           (COND  ((ATOM REM)
                        (ERROR     "Argument von OUR-REVERSE ~
                                   keine korrekte Liste: ~S"  LST)))))
OUR-REVERSE
```

Hier haben wir die Funktion ERROR eingesetzt; sie hat das
Aufrufmuster:

(ERROR <Steuerstring> <Arg-1> ... <Arg-n>)

Dieses erinnert uns an das **Aufrufmuster** von Format :

(FORMAT <Strom> <Steuerstring> <Arg-1> ... <Arg-n>)

Die auffallende Ähnlichkeit ist kein Zufall; die ERROR-Form wird wie folgt abgearbeitet:

- Zuerst wird das Symbol ERROR als Hinweis ausgegeben.
- Dann werden Steuerstring und nachfolgende Argumente an FORMAT weitergereicht. Unter Verwendung dieser Argumente und einer impliziten Spezifikation von Strom, welche die Ausgabe auf das Benutzerterminal führt, wird FORMAT abgearbeitet; dabei kommt es zu einer Ausgabe, die in einer durch den Steuerstring und die Argumente festgelegten Weise den Fehler erläutert.
- Im allgemeinen wird gemeldet, aus welcher Funktion die Fehlermeldung kam.
- Anschließend wird die Abarbeitung unterbrochen und implementationsabhängig der sogenannte Debugger gerufen, das ist ein Systemprogramm, das die Fehlererkennung und Fehlerbehandlung unterstützt. Wir gehen darauf in Kapitel 22 etwas näher ein. Doch ist die Art der Fehlerbehandlung in COMMON LISP selbst nicht festgelegt und somit implementationsabhängig.

Beispiel:

Wir kommen zurück zur Definition von OUR-REVERSE einschließlich des Programmstücks zur Fehlermeldung.

```
-->(DEFUN OUR-REVERSE (LST)
    (DO  ((REM LST (CDR REM))
          (AUX 'NIL (CONS (CAR REM) AUX)))
         ((NULL REM) AUX)
                                        ; Rumpf der DO-Form war leer
       (COND  ((ATOM REM)
               (ERROR "Argument von OUR-REVERSE ~
               keine korrekte Liste: ~S"  LST)))))
OUR-REVERSE
```

Jetzt werden alle Fehler aufgrund unkorrekter Argumente gemeldet.

Beispiele:

```
---> (OUR-REVERSE '(A B C))
(C B A)                          ; Liste wird invertiert

---> (OUR-REVERSE 'A)
>> ERROR: Argument von OUR-REVERSE keine korrekte Liste: A
   ERROR signalled by function OUR-REVERSE
>> quit
---> (OUR-REVERSE '(A B . C)
>> ERROR: Argument von OUR-REVERSE keine korrekte Liste: (A B . C)
   ERROR signalled by function OUR-REVERSE
>> quit
```

13.6 Zusammenfassung

Formatierte Ausgabe erstellt man am besten mit FORMAT; von ihren vielen Möglichkeiten konnte hier nur ein kleiner Ausschnitt vorgestellt werden.

Mit ERROR können die Fähigkeiten von FORMAT für die Ausgabe von Fehlerdiagnosen eingesetzt werden.

Fehlermeldungen mit ERROR gehen immer einher mit einem Abbruch der Programmauswertung. Es gibt auch Fehlerdiagnosefunktionen, die nach Erkennen und Korrektur des Fehlers eine Fortführung der Auswertung ermöglichen.

KAPITEL 14:

Parameterspezifikationen beim Funktionskonzept

Wir haben eine größere Anzahl von Systemfunktionen kennengelernt und Erfahrungen mit der Definition von "eigenen" Funktionen gewonnen.
Unter den Systemfunktionen waren einige (z.B. READ und PRINT), die optionale Argumente haben. Man nennt ein Argument **optional**, wenn der Programmierer das Recht hat zu entscheiden (so etwas nennt man Option), ob er diese Argumente beim Aufruf explizit angeben will oder nicht.
Andere Systemfunktionen (z.B. + und APPEND) können nach Wahl des Programmierers mit null, einem oder mehr Argumenten aufgerufen werden, haben also beliebig viele Argumente.

Wieder andere Systemfunktionen können optional mit sogenannten **Schlüsselwort-Argumenten** aufgerufen werden. Ein Beispiel ist die Funktion MEMBER, bei der wir in Aufrufen der Gestalt

(MEMBER <Ausdruck> <-Liste-> :TEST <Testprädikat>)

ein von EQ abweichendes Testprädikat festlegen können.

All diese Möglichkeiten gibt es auch bei selbstdefinierten Funktionen. Wir wollen sie in dieser Lektion kennenlernen.

14.1 Optionale Argumente

Wir wissen schon lange, wie wir mit Hilfe von Lambda-Ausdrücken und DEFUN-Formen selbst Funktionen definieren können. Eine derartige Funktionsdefinition hat die Gestalt

(LAMBDA <Parameterliste> <Funktionsrumpf>)

bzw.

(DEFUN <Funktionsname> <Parameterliste> <Funktionsrumpf>).

Dabei ist als Parameterliste bisher nur eine Liste der Gestalt

(<Par-1> <Par-2> ... <Par-n>)

mit n ≥ 0 Symbolen Par-1 bis Par-n als Parameter zugelassen.
Im Aufruf einer so definierten Funktion mußte für jeden Parameter ein Argument zur Verfügung stehen. Parameterlisten der obigen Gestalt spezifizieren demnach alle Argumente als obligatorisch, d.h. als verpflichtend.

Will man bestimmte Argumente als optional spezifizieren und damit bei jedem Aufruf (neu !) wählen können, ob man diese Argumente angeben will oder nicht, so muß dies bereits in der Parameterliste festgelegt werden.
Dies kann geschehen mit einer Parameterliste der Gestalt

(<Par-1> ... <Par-n> &OPTIONAL <SOPar-1> ... <SOPar-m>)

Hier sind Par-1 bis Par-n Parameter für obligatorische Argumente; das Symbol &OPTIONAL ist kein Parameter, sondern ein **"Parameterlisten-Schlüsselwort"**; es leitet in der Parameterliste die Spezifikationen der Parameter für optionale Argumente ein.

SOPar-1, ... SOPar-m stehen für derartige Spezifikationen von Parametern für optionale Argumente.

Diese Spezifikationen haben häufig die Gestalt

(<Symbol> <Default-Argument>)

mit einem Symbol als Parameter und einem Ausdruck als Default-Argument. Die Bezeichnung Default-Argument bedeutet, daß dieser Ausdruck dem Parameter als Argument zugeordnet wird, falls ein Funktionsaufruf für diesen Parameter kein Argument bereitstellt.

Der Aufruf einer Funktion mit n obligatorischen und m optionalen Argumenten enthält mindestens n und höchstens n+m Argumente. Zur Auswertung des Aufrufs werden alle Argumente ausgewertet. Zuerst werden die n Parameter für obligatorische Argumente an die ersten n Argumentwerte gebunden. An die noch verbleibenden Argumentwerte werden dann der Reihe nach die Parameter für optionale Argumente gebunden. Jeder Parameter, der bei dieser Zuordnung von Argumenten aus dem Funktionsaufruf leer ausgeht, wird an den Wert des ihm

zugehörigen Default-Arguments gebunden. Unter diesen Bindungen wird dann der Funktionsrumpf ausgewertet.

Beispiel 1:

Im folgenden definieren wir ein Prädikat ETWA-GLEICH, das zwei Zahlen X und Y dann als "etwa-gleich" ansieht, wenn der Unterschied einen bestimmten Toleranzwert nicht überschreitet. Dieser Toleranzwert wird als optionales Argument eingegeben; fehlt das Argument für den Toleranzwert, so wird der in der Funktionsdefinition festgelegte Default-Wert zugrundegelegt.

```
-->(DEFUN ETWA-GLEICH (X Y &OPTIONAL (TOLERANZ 0.001))
     (< (ABS (- X Y)) TOLERANZ))          ; das Prädikat < testet, ob der
ETWA-GLEICH                               ; Unterschied zwischen X und Y
                                          ; kleiner ist als der Wert von
                                          ; TOLERANZ

-->(ETWA-GLEICH 0.1614 0.1617)
T
-->(ETWA-GLEICH 0.1614 0.1617 0.0002)
NIL
-->(ETWA-GLEICH 0.162 0.164)
NIL
```

Eine Spezifikation für einen optionalen Parameter mit einem Default-Argument gleich NIL, also der Gestalt

(<Symbol> NIL) **kann abgekürzt werden zu** <Symbol> .

Beispiel 2:

```
--> (DEFUN BEISPIEL (A &OPTIONAL B) (AND A B))
BEISPIEL
--> (BEISPIEL 17 4)
4
--> (BEISPIEL 17)
NIL                                       ; hier wurde für B der Defaultwert
                                          ; zugrundegelegt
```

Soll die Auswertung einer Funktion mit optionalen Argumenten auch davon abhängen, ob im zugrundeliegenden Funktionsaufruf für einen bestimmten Parameter ein Argument zur Verfügung stand oder nicht (und deshalb der Default-Wert herangezogen werden mußte), so verwendet man Spezifikationen der Gestalt

(<Parameter> <Default-Argument> <SP-Parameter>)

Hier steht SP-Parameter für "Supplied-Property-Parameter". Dieser SP-Parameter ist ein Symbol; er wird für die Dauer der Auswertung des jeweiligen Funktionsaufrufs an den Wert T gebunden, falls im Funktionsaufruf für den zugrundeliegenden Parameter ein Argument zur Verfügung stand; er wird an den Wert NIL gebunden, falls kein Argument zur Verfügung stand und deshalb der Default-Wert für die Auswertung zugrundegelegt werden mußte.

Beispiel:

```
-->(DEFUN BEISPIEL (&OPTIONAL (A 3 B) (C 4 D))
        (LIST A B C D))                    ; B und D sind SP-Parameter
BEISPIEL

-->(BEISPIEL 17)
(17 T 4 NIL)                               ; B wurde an T, C an Defaultwert 4
                                           ; und D an NIL gebunden

-->(BEISPIEL 17 5)
(17 T 5 T)                                 ; B und D wurden an T gebunden

-->(BEISPIEL)
(3 NIL 4 NIL)
```

14.2 Beliebig viele Argumente

Systemfunktionen wie + oder APPEND und LIST können mit beliebig vielen Argumenten aufgerufen werden und sind deshalb sehr flexibel einsetzbar.
Eine ähnliche Flexibilität erreicht man bei selbstdefinierten Funktionen durch Spezifikation eines sogenannten Restparameters. Wir nehmen an, die Definition einer selbstdefinierten Funktion enthält eine Parameterliste der Gestalt

```
(<Par-1> ... <Par-n>  &REST <Restpar>)
oder
(<Par-1> ... <Par-n>  &OPTIONAL <SOPar-1> ... <SOPar-m>
                      &REST <Restpar>)
```

Durch das Parameterlisten-Schlüsselwort &REST wird das unmittelbar folgende Symbol als sogenannter Restparameter spezifiziert.
Im Aufruf einer so definierten Funktion können nach den obligatorischen Argumenten für die Parameter Par-1 bis Par-n noch beliebig viele weitere Argumente auftreten. Bei der Auswertung des Aufrufs werden alle vorhandenen Argumente ausgewertet und ihre Werte der Reihe nach zunächst den obligatorischen Parametern und dann den eventuell vorhandenen optionalen Parametern zugeordnet. Sind nach Versorgung aller obligatorischen und optionalen Parameter noch Argumentwerte übrig, so werden diese in einer Liste zusammengefaßt, die dem Restparameter zugeordnet wird.
Sind keine Argumentwerte mehr übrig, so wird dem Restparameter die leere Liste zugeordnet. Unter diesen Bindungen wird dann der Funktionsrumpf ausgewertet.

Beispiele:

```
-->(DEFUN BEISPIEL (A &OPTIONAL (B 2))
     (+ A (* B 3)))
BEISPIEL

-->(BEISPIEL 4)                          ; A wurde an 4 und B an Defaultwert
10                                       ; 2 gebunden

-->(BEISPIEL 4 5)                        ; A wurde an 4 und B an 5 gebunden
19


-->(DEFUN EXAMPLE (A &OPTIONAL (B 2) (C 3) &REST X)
     (LIST A B C X))
EXAMPLE

-->(EXAMPLE 6 3 8 9 10 11)               ; A wurde an 6, B an 3, C an 8 und X
(6 3 8 (9 10 11))                        ; an die Liste (9 10 11)
                                         ; gebunden
```

```
-->(EXAMPLE 6 3 8)                ; hier wird X an die leere Liste,
(6 3 8 NIL)                       ; also an NIL gebunden

-->(EXAMPLE 6 3)                  ; hier wird A an 6, B an 3, C an
(6 3 3 NIL)                       ; Defaultwert 3 und X an NIL
                                  ; gebunden
```

14.3 Schlüsselwort-Argumente, ein anderes Konzept für optionale Argumente

Wir haben einige Systemfunktionen gesehen, bei denen im Aufruf sogenannte Schlüsselwort-Argumente verwendet werden können.

Ein **Beispiel** ist die Funktion MEMBER:
Bei Abarbeitung eines Aufrufs der Gestalt

(MEMBER <Ausdruck> <-Liste->)

wird für den Vergleich des Wertes des 1. Arguments mit den Elementen der Liste das Prädikat EQL verwendet.

Dagegen bewirkt ein Aufruf der Gestalt

(MEMBER <Ausdruck> <-Liste-> :TEST <2-stelliges Prädikat>) ,

daß für den Vergleich das in der 4. Argumentposition angegebene Prädikat herangezogen wird.

Die in der 3. Argumentsposition angegebene Zeichenfolge :TEST ist durch den einleitenden Doppelpunkt als Schlüselwort gekennzeichnet. Das Auftreten von :TEST in diesem Aufruf hat nur den Zweck darauf hinzuweisen, daß für den optionalen Parameter TEST von MEMBER ein Argument angegeben werden wird. Dieses Argument wird dann auch unmittelbar nach dem den Parameter TEST bezeichnenden Schlüsselwort :TEST angegeben.

Ein Schlüsselwort wie :TEST, das im Aufruf einer Funktion in Argumentposition auftritt, bezeichnen wir der Einfachheit halber auch als Argument. Da es aber i.a. keinen Wert einbringen soll, ist festgelegt, daß Schlüsselworte - das sind Symbole, die mit einem Doppelpunkt beginnen - stets sich selbst als Wert haben.

Ein anderes Beispiel ist die Funktion LOAD:
Der Aufruf (LOAD "FOO.LISP") mit dem einzigen Argument "FOO.LISP" bewirkt, daß das durch den String "FOO.LISP" bezeichnete File "geladen" wird, das heißt, daß die im File stehenden Ausdrücke - typischerweise Funktionsdefinitionen und Wertzuweisungen - der Reihe nach eingelesen und ausgewertet werden, wobei die Werte dieser Ausdrücke nicht ausgegeben werden.

Dagegen bewirkt der Aufruf

(LOAD "F00.LISP" :PRINT T)

daß zusätzlich der Wert jedes ausgewerteten Ausdrucks über die Standard-Ausgabe ausgegeben wird.

Das zweite Argument im obigen Ausruf ist das Schlüsselwort :PRINT. Sein Auftreten in diesem Aufruf von LOAD hat nur den Zweck darauf hinzuweisen, daß für den optionalen Parameter PRINT in diesem Aufruf ein Argument angegeben werden wird; dieses Argument - hier T - folgt wieder unmittelbar auf das den Parameter bezeichnende Schlüsselwort.

Mit dem Schlüsselwort-Argument verfügen wir letztlich über ein zweites Konzept für optionale Argumente:

- Beim ersten Konzept - es wurde im vorhergehenden Kapitel beschrieben - werden die Parameter in der Reihenfolge ihres Auftretens in der Parameterliste solange mit Argumenten versorgt, wie Argumente vorhanden sind.

- Beim Konzept der Schlüsselwort-Argumente werden in jedem Aufruf diejenigen Parameter, für die Argumente angegeben werden sollen, jeweils explizit mit den zugehörigen Argumenten angegeben. Dieses Konzept ermöglicht es, für eine Funktion eine größere Anzahl von optionalen Argumenten vorzusehen ohne dabei gezwungen zu sein, diese im Aufruf in einer bestimmten Reihenfolge anzugeben. Dies ist besonders dann von Vorteil, wenn in Aufrufen häufig nur ein Bruchteil dieser Argumente tatsächlich gebraucht wird.

Bei beiden Konzepten tritt als Ersatz für ein im Aufruf nicht geliefertes Argument jeweils ein festgelegter Default-Wert ein.

Die Schlüsselworte wie :PRINT, :DIRECTION, :INPUT und dergleichen sind grundsätzlich zu unterscheiden von den Parameterlisten-Schlüsselworten wie &OPTIONAL oder &REST. Letztere treten nur in der Parameterspezifikation, niemals in Argumentsposition eines Aufrufs auf und haben undefinierte Werte.

Schlüsselwort-Argumente können nun auch bei selbstdefinierten Funktionen eingeführt werden:

Wir gehen aus von einer Parameterliste der Gestalt

(<Par-1> ... <Par-n> &KEY <KPar-1> ... <KPar-r>)

Das Parameterlisten-Schlüsselwort &KEY leitet hier eine Folge von $r \geq 0$ Parameterspezifikationen für Schlüsselwort-Argumente ein. Wir betrachten zunächst Spezifikationen der Gestalt

(<Parameter> <Default-Argument>)
und
<Parameter> als Abkürzung für obige Liste im Falle des Default-Arguments NIL

In Funktionsaufrufen können dann für beliebige der so mit &KEY spezifizierten Parameter Argumente angegeben werden. Allerdings muß jetzt jeder Parameter, der ein Argument erhalten soll, zusammen mit diesem Argument angegeben werden. Beide Angaben erfolgen in Argumentsposition: Auf ein Argument das einen Parameter identifiziert, folgt jeweils das diesem Parameter zugeordnete Argument.

Das Argument, das einen Parameter identifiziert, besteht im Fall einer Parameterspezifikation wie oben aus der Parameterbezeichnung mit vorangestelltem Doppelpunkt. Jede Zeichenfolge, die von einem Doppelpunkt eingeleitet wird, nennt man **Schlüsselwort.**

Jedes Schlüsselwort hat sich selbst als Wert; ein Argument, das aus einem Schlüsselwort besteht und einen Parameter identifizert, wird also nicht mehr weiter ausgewertet.

Beispiele für solche Schlüsselwörter haben wir bei einigen Systemfunktionen kennengelernt. Diese Schlüsselworte, wie :PRINT und :TEST, die letztlich Parameter bezeichnen, denen Argumente zugeordnet werden können, sind zu unterscheiden von den Spezifikations-Schlüsselwörter &OPTIONAL, &REST und &KEY, die bei der Parameterspezifikation von Funktionen nicht aber in deren Aufruf verwendet werden.

Beispiele:

```
-->(DEFUN BEISPIEL (A B &KEY C (D A))
     (LIST A B C D))                    ; C hat Default-Argument NIL
BEISPIEL                                ; D hat Default-Argument A

-->(BEISPIEL 1 2)                       ; Zuerst werden die obligatorischen
(1 2 NIL 1)                             ; Parameter A und B an ihre
                                        ; Argumentwerte 1 und 2 gebunden;
                                        ; danach erst werden die
                                        ; Schlüsselwort-Argumente ausge-
                                        ; wertet; der Default-Wert für D ist
                                        ; gleich dem Wert des Parameters A,
                                        ; also gleich 1

-->(BEISPIEL 1 2 :C 6)
(1 2 6 1)

-->(BEISPIEL 1 2 :D 8)
(1 2 NIL 8)

-->(BEISPIEL 1 2 :C 6 :D 8)
(1 2 6 8)

-->(BEISPIEL 1 2 :D 8 :C 6)             ; Reihenfolge bei Schlüsselwort-Argu-
(1 2 6 8)                               ; menten unwesentlich


--> (BEISPIEL :A 1 :D 8 :C 6)
(:A 1 6 8)

-->(BEISPIEL :A :B :C :D)               ; A wird an :A, B an :B, C an
(:A :B :D :A)                           ; :D und D an seinen Default-Wert,
                                        ; also an den Wert von A gebunden.
```

Auch die Auswertung einer Funktion mit Schlüsselwort-Argumenten kann davon abhängig gemacht werden, ob im zugrundeliegenden Funktionsaufruf für einen bestimmten Parameter ein Argument zur Verfügung stand oder nicht (und deshalb der Defaultwert herangezogen werden mußte). Die Spezifikation hat dieselbe Gestalt und dieselbe Auswirkung wie bei gewöhnlichen optionalen Parametern.

14.4 Zusammenspiel verschiedener Parameterspezifikationen

Die oben betrachtete Funktion BEISPIEL enthielt nur Parameter für obligatorische Argumente und Schlüsselwort-Argumente. Tatsächlich kann eine selbstdefinierte Funktion nebeneinander Parameter für obligatorische Argumente und normale optionale Argumente, einen Restparameter sowie Parameter für Schlüsselwort-Argumente enthalten. Für die Reihenfolge der Parameterspezifikationen in der Parameterliste gilt dabei:
Zuerst sind die Parameter für obligatorische Argumente aufzulisten, dann hinter dem Schlüsselwort &OPTIONAL alle Parameterspezifikationen für normale optionale Argumente, dann hinter dem Schlüsselwort &REST der Restparameter, dann hinter dem Schlüsselwort &KEY die Parameterspezifikationen für optionale Schlüsselwortargumente.

Bei Aufruf und Auswertung einer Funktion werden zuerst die Parameter für obligatorische Argumente an Argumentwerte gebunden, dann soweit möglich die Parameter für optionale Argumente.

Hat die Funktion keinen Restparameter und keinen Schlüsselwort-Parameter, so dürfen jetzt keine Argumente übrig sein, sonst liegt ein Fehler vor.

Hat die Funktion einen Restparameter, so wird dieser an die Liste der bis jetzt nicht verbrauchten Argumentwerte gebunden. Sind keine Argumentwerte übrig, so wird der Restparameter an NIL gebunden.

Hat die Funktion einen oder mehrere Schlüsselwort-Parameter aber keinen Restparameter, so muß die Anzahl der verbliebenen Argumente gerade sein; diese werden dann in Paaren zusammengefaßt, wobei die erste Komponente jeweils ein Schlüsselwort sein muß und die zweite als zugehöriger Argumentwert interpretiert wird.

Sind sowohl ein Restparameter als auch Parameter für Schlüsselwort-Argumente vorhanden, so werden die dem Restparameter bereits zugeordneten Argumentwerte nochmals für die

Schlüsselwort-Parameter verwendet. Ihre Anzahl muß wieder gerade sein, damit sie zu Paaren zusammengefaßt werden können, deren erste Komponente ein Schlüsselwort sein muß und deren zweite Komponente als zugehöriger Argumentwert interpretiert wird.

Das ist übrigens der einzige Fall, in dem ein Argument für verschiedene Parameter verwendet werden kann.

Beispiele:

```
-->(DEFUN BEISPIEL (A &OPTIONAL (B 3)
                        &REST X &KEY C (D A))
        (LIST A B C D X)
BEISPIEL

-->(BEISPIEL 1)
(1 3 NIL 1 NIL)

-->(BEISPIEL 1 2)
(1 2 NIL 1 NIL)

-->(BEISPIEL :C 7)
(:C 7 NIL :C NIL)

-->(BEISPIEL 1 6 :C 7)
(1 6 7 1 (:C 7))

-->(BEISPIEL 1 6 :D 8)
(1 6 NIL 8 (:D 8))

-->(BEISPIEL 1 6 :D 8 :C 9 :D 10)
(1 6 9 8 (:D 8 :C 9 :D 10))
```

14.5 Zusammenfassung

Wir haben in dieser Lektion gelernt, wie wir selbst Funktionen mit optionalen Argumenten und/oder mit beliebig vielen Argumenten definieren können.
Für optionale Argumente haben wir zwei Konzepte kennengelernt:

- Normale optionale Argumente und
- Schlüsselwort-Argumente.

Für normale optionale Argumente werden Spezifikationen in der Parameterliste der Funktionsdefinition hinter dem Parameterlisten-Schlüsselwort &OPTIONAL aufgelistet; die Zuordnung von Argumenten an Parameter wird durch ihre Reihenfolge und Verfügbarkeit geregelt.

Für Schlüsselwort-Argumente werden die Parameterspezifikationen in der Parameterliste mit dem Parameterlisten-Schlüsselwort &KEY eingeleitet; hier muß zur Zuordnung von Argumenten im Funktionsaufruf jeder Parameter, dem ein Argument explizit zugeordnet werden soll, auch explizit vor diesem Argument genannt werden; dabei wird ihm ein Doppelpunkt vorangestellt.

Wird für einen optionalen Parameter im Funktionsaufruf kein Argument angegeben, so wird automatisch ein Default-Wert herangezogen. Ist dieser nicht explizit festgelegt, so dient NIL als Default-Wert.

Soll eine Funktion beliebig viele Argumente haben können, so definiert man einen sogenannten Restparameter. Er wird in der Parameterspezifikation durch Voranstellen des Parameterlisten-Schlüsselworts &REST gekennzeichnet. Beim Aufruf übernimmt dieser Restparameter die nach Versorgung von obligatorischen und normalen optionalen Parametern noch verbleibenden Argumente in einer Liste.

KAPITEL 15:

Makros

Allgemein versteht man unter einem Makro ein Symbol, das stellvertretend für ein in der Regel längeres Programmstück verwendet wird (Makro bedeutet "etwas Großes"). Dieses längere Programmstück wird ermittelt durch die sogenannte Makro-Expansion und dann anstelle des Makros ausgewertet. Ein Makro hat im allgemeinen Parameter, die diese Expansion steuern oder in dem durch Expansion gewonnenen Programmstück verwendet werden.

Mit Hilfe von Makros kann man LISP-Formen definieren, die nicht unmittelbar ausgewertet, sondern in andere LISP-Formen überführt, wir sagen auch expandiert werden, welche dann anstelle der ursprünglich gegebenen Formen ausgewertet werden.

Der Programmierer hat so die Möglichkeit, für längere und häufig wiederkehrende Folgen von Befehlen bzw. Funktionsanwendungen abgekürzte Schreibweisen einzuführen.

Wir haben festgestellt, daß Programme und Daten in LISP im Prinzip dieselbe Gestalt haben. Programme können deshalb auch genauso wie Daten manipuliert werden. Die bei der Verarbeitung eines Makros durchzuführende Expansion eines Programmstücks kann deshalb in einfacher Weise direkt in LISP beschrieben werden.

15.1 Zur Definition von Makros

Makros werden definiert mit Hilfe der speziellen Form DEFMACRO, diese ist selbst ein Makro. Eine Makrodefinition hat die Gestalt

(DEFMACRO <Makroname> <Parameterliste> <Makrorumpf>) ,
sieht also ähnlich aus wie eine Funktionsdefinition mit DEFUN.

Wert

einer Makro-Definition mit DEFMACRO ist wie bei DEFUN der Name des Makros. Auch die DEFMACRO-Form wird nicht wegen dieses Wertes, sondern wegen ihres Seiteneffekts ausgewertet. Dieser ordnet der Makrobezeichnung den Rumpf als Expansionsfunktion zu.

Ein so definiertes Makro kann aufgerufen werden mit einem Makro-Aufruf der Gestalt

(<Makroname> <Argumentliste>)

Wird in einer auszuwertenden Form als erstes Listenelement der Name eines Makros entdeckt, so werden ohne weitere Auswertung alle restlichen Listenelemente an den Makrorumpf übergeben. Mit anderen Worten: Alle im Makrorumpf auftretenden Parameter werden an die nicht ausgewerteten Argumente gebunden. Der Makrorumpf beschreibt die an Stelle des Makroaufrufs auszuwertende Form. Er bestimmt so die Makroexpansion des Makroaufrufs. Wenn die durch einen Expansionsschritt gewonnene Form als erstes Listenelement wiederum eine Makrobezeichnung enthält, wird erneut eine Makro-Expansion durchgeführt. Dieser Prozeß der Makroexpansion wird solange fortgeführt, bis in der ersten Position der resultierenden Liste kein Makro mehr erscheint. Sodann wird dieses Ergebnis der Makroexpansion an Stelle des Makroaufrufs ausgewertet.

Wir betrachten im folgenden Kapitel ein Standardbeispiel .

15.2 Das Makro IF

Ein Programmierer, der häufig Bedingungen der folgenden einfachen Gestalt

```
(COND  (<Test> <Ausdruck bei erfolgreichem Test>)
       (T      <Ausdruck bei erfolglosem Test>))
```

hat, wünscht sich eine übersichtlichere und bequemer zu handhabende Sprachkonstruktion. Er denkt an eine "IF-Form" mit einem Aufrufmuster der Gestalt

```
(IF <Test>    <Ausdruck bei erfolgreichem Test>
              <Ausdruck bei erfolglosem Test>)
```

Diese läßt sich allerdings mit DEFUN kaum definieren; ein Versuch mit

```
(DEFUN TRIAL-IF (TEST THEN ELSE)
       (COND    (TEST THEN)
                (T  ELSE)))
```

liefert sicher eine Funktion, die andere Eigenschaften hat als obige COND-Form.

Insbesondere werden zu jedem Aufruf
(TRIAL-IF <Test> <Ausdruck-1> <Ausdruck-2>)

als erstes alle Argumente ausgewertet. Die COND-Form im Rumpf benötigt die Argumente aber unausgewertet.

Die folgende Definition eines Makros OUR-IF erfüllt jedoch unsere Wünsche.

```
-->(DEFMACRO OUR-IF (TEST THEN ELSE)
       (LIST   'COND
                    (LIST TEST THEN)
                    (LIST 'T ELSE)))
OUR-IF
```

Die Parameterliste enthält 3 Parameter nämlich TEST, THEN und ELSE für obligatorische Argumente.

Ein Aufruf des Makros OUR-IF hat jetzt die Gestalt

(OUR-IF <Test> <Ausdruck-1> <Ausdruck-2>)

Durch Makro-Expansion entsteht hieraus die Form

```
(COND  (<Test> <Ausdruck-1>)
       (T      <Ausdruck-2>))  ,
```

die nun an Stelle der OUR-IF-Form ausgewertet wird.

Beispiel:

```
-->(SETQ X 4)
4
-->(SETQ Y 5)
5
-->(OUR-IF (< X Y)    (SETQ A X)
                      (SETQ B Y))
4
-->A
4
-->B
ERROR: B has no global value.
```

Ganz anders würde die Auswertung der oben definierten Funktion TRIAL-IF mit denselben Argumenten ablaufen. Bei Aufruf von

```
(TRIAL-IF (< X Y)   (SETQ A X)
                    (SETQ B Y))
```

werden zunächst alle Argumente abgearbeitet und dabei insbesondere auch beide Wertzuweisungen ausgeführt.

Wir betrachten die Auswertung des obigen Makroaufrufs genauer:

```
-->(OUR-IF (< X Y)  (SETQ A X)
                    (SETQ B Y))
4
```

Der Interpretierer, d.h. die Funktion EVAL ruft nach Erkennen des Listenelements OUR-IF als Bezeichung für ein Makro intern die Funktion MACROEXPAND, wertet nach Durchführung der Makro-Expansion das dadurch erhaltene Programmstück aus und übergibt das ermittelte Resultat als Wert des ursprünglichen Makroaufrufs.

Die Funktion MACROEXPAND kann aber auch explizit aufgerufen werden.

Ein solcher Aufruf hat im einfachsten Fall die Gestalt

(MACROEXPAND <Form>)

er liefert stets zwei Werte, nämlich

- das Ergebnis der Makroexpansion und T, wenn die Form einen Makroaufruf darstellt
- die Form selbst und NIL, wenn die Form keinen Makroaufruf darstellt.

Beispiele:

```
-->(MACROEXPAND '(OUR-IF   (< X Y)
                           (SETQ A X)
                           (SETQ B Y)))
(COND ((< X Y)  (SETQ A X))
      (T        (SETQ B Y)))

T

-->(MAKROEXPAND '(TRIAL-IF (< X Y)
                           (SETQ A X)
                           (SETQ B Y)))
(TRIAL-IF    (< X Y)
             (SETQ A X)
             (SETQ B Y)))

NIL
```

Die Makroexpansion liefert also für OUR-IF gerade die COND-Form, von der unsere Überlegungen ausgingen. Wir weisen aber darauf hin, daß COND-Formen dieser einfachen Gestalt in der Ausgabe häufig
durch eine äquivalente IF-Form ersetzt werden.

Die explizite Bestimmung der Makro-Expansion ist vor allem bei komplexeren Makros für die Fehlersuche interessant.

Wir wollen das Makro OUR-IF noch etwas flexibler gestalten: Es soll optional entweder wie bisher mit zwei Alternativen - für positives und negatives Testresultat - oder mit einer Alternative -

nur für positives Testresultat - benutzt werden können. Dazu spezifizieren wir die Parameter TEST und THEN als obligatorisch, den Parameter ELSE jedoch als optional mit dem DEFAULT- Wert NIL.

```
(DEFMACRO OUR-IF (TEST THEN &OPTIONAL (ELSE 'NIL))
        (LIST 'COND
                        (LIST TEST THEN)
                        (LIST 'T  ELSE)))
```

15.3 Makros für spezielle Formen

Formen, die nicht jedes ihrer Argumente auswerten, hatten wir als spezielle Formen bezeichnet. Jetzt haben wir festgestellt, daß bei Expansion eines Makroaufrufs die Parameter an die unausgewerteten Argumente gebunden werden. Mit dem Makro-Konzept können wir also auch selbst spezielle Formen definieren. Ein Beispiel dafür ist bereits das Makro IF.
Wir versuchen nun, ein Makro für die LET-Form zu schreiben. Es muß einen LET-Aufruf auf die Auswertung eines LAMBDA-Ausdrucks zurückführen. Das bedeutet, jede LET-Form der Gestalt

```
(LET    ((<Parameter-1> <Argument-1>)
        .
        .
        .
        (<Parameter-n> <Argument-n>))
        <Ausdruck-1>  .....                <Ausdruck-m>)
```

wird als Makroaufruf eines Makros LET aufgefaßt, aus dem durch Makroexpansion zuerst eine äquivalente LAMBDA-Anwendung gewonnen wird, die dann anstelle der ursprünglichen LET-Form ausgewertet wird.
Zur Definition dieses Makros - wir nennen es OUR-LET - verwenden wir den obligatorischen Parameter PARARG und den Restparameter BODY. Beim Aufruf von LET soll PARARG an die Liste der Parameter-Argument-Paare und BODY an die Liste der Ausdrücke des Rumpfs von LET gebunden werden.

Die Definition für ein Makro OUR-LET hat damit die Gestalt

```
(DEFMACRO OUR-LET (PARARG &REST BODY)
        <Rumpf der Makrodefinition>)
```

Der Rumpf der Makrodefinition wird jetzt so konstruiert, daß seine Auswertung den Ausdruck für die folgende Lambda-Anwendung liefert.

```
((LAMBDA      (<Par-1>  ...      <Par-n>)
              <Ausdruck-1>  ...  <Ausdruck-m>)
              <Arg-1>  ...       <Arg-n>)
```

Wir konstruieren zuerst die Parameterliste; man erhält sie aus den ersten Komponenten der Parameter-Argument-Paare, also aus den Elementen der PARARG zugeordneten Liste durch Zusammenfassen mit LIST. Sie ist also gegeben durch den Wert von

```
(MAPCAR #'CAR PARARG)
```

Jetzt zum Lambda-Ausdruck: Sein Rumpf ist identisch mit dem Rumpf des LET-Ausdrucks. Allerdings sind die Ausdrücke im Rumpf des LET-Ausdrucks für die Zuordnung an den Restparameter BODY zu einer Liste zusammenzufassen. Wenn man an diese Liste mit LIST* vorne als zusätzliche Elemente zunächst die Parameterliste und dann das Symbol LAMBDA anfügt, ergibt sich gerade der Lambda-Ausdruck: Er ist somit gegeben durch den Wert der Form

```
(LIST* 'LAMBDA (MAPCAR #'CAR PARARG) BODY)
```

Jetzt konstruieren wir die Liste der Argumente; man erhält sie ähnlich wie die Parameterliste aus den zweiten Komponenten der Parameter-Argument-Paare durch Zusammenfassen mit LIST. Sie ist also gegeben durch den Wert von

```
(MAPCAR #'CADR PARARG)
```

Die Lambda-Anwendung erhält man jetzt insgesamt durch Auswerten von

```
(LIST* <Lambda-Ausdruck> <Argumentliste>)
```

Diese LIST*-Form bildet somit den Rumpf der Makrodefinition. Die Makrodefinition selbst lautet

Diese LIST*-Form bildet somit den Rumpf der Makrodefinition. Die Makrodefinition selbst lautet jetzt

```
(DEFMACRO OUR-LET (PARARG &REST BODY)
   (LIST* (LIST*   'LAMBDA
                   (MAPCAR #'CAR PARARG)
                   BODY)
          (MAPCAR #'CADR PARARG)))
```

Nach Auswertung dieser Makrodefinition können wir das Makro OUR-LET anwenden.

Beispiel:

```
->(OUR-LET   ((A 4
               (B 'SALE))
              (LIST 'CAR A B))
(CAR 4 SALE)
```

Wir betrachten zur Illustration noch die zugehörige Makro-Expansion

```
->(MACROEXPAND '(OUR-LET ((A 4)
                          (B 'SALE))
                          (LIST 'CAR A B)))
((LAMBDA (A B) (LIST 'CAR A B))
         4 'SALE)

T
```

15.4 Backquote

Der Rumpf eines Makros beschreibt die Makroexpansion, also die Konstruktion eines Programmstücks. Das Ergebnis der Konstruktion ist beim Rumpf unseres Makros OUR-LET noch einigermaßen gut ablesbar. Leichter erkennbar wäre es sicherlich, wenn wir als Rumpf einfach ein Muster des zu erzeugenden Programmstücks angeben könnten, etwa folgender Art

Muster ((LAMBDA
<Wert von (MAPCAR #'CAR PARARG)>
<Elemente der Liste, die Wert von BODY>)
<Elemente der Liste, die Wert von (MAPCAR #'CADR PARARG)>)

In diesem Muster soll nur dort ausgewertet werden, wo dies ausdrücklich gefordert wird.

Einen Ausdruck, der als Muster angesehen werden soll, kennzeichnet man durch Voranstellen des Zeichens ` (gesprochen "Backquote").
Dieses "Backquote" wirkt zunächst wie QUOTE, indem es die Auswertung des nachfolgenden Ausdrucks verhindert. Doch kann bei dieser "Backquotierung" für Teilausdrücke wieder Auswertung angeordnet werden:
So bewirkt Voranstellen eines Kommas vor einem Teilausdruck, daß dieser Teilausdruck ausgewertet wird und sein Wert an Stelle des Teilausdrucks eingesetzt wird.
Das Voranstellen der Zeichenkombination ,@ also von Komma und nachfolgendem at-Zeichen vor einem Teilausdruck bewirkt, daß der betreffende Teilausdruck ausgewertet wird und die Elemente der dabei resultierenden Liste, aber ohne das äußere Klammerpaar, an Stelle des Teilausdrucks eingefügt werden.

Das beschriebene Backquote-Konzept erweist sich in der Tat als sehr hilfreich bei der Definition von Makros. Wir können damit zum Beispiel die Definition des Makros OUR-LET wie folgt vornehmen:

```
(DEFMACRO OUR-LET (PARARG &REST BODY)
    `((LAMBDA      ,(MAPCAR #'CAR PARARG)
                   ,@BODY)
    ,@(MAPCAR #'CADR PARARG))
)
```

Der Rumpf zeigt jetzt viel deutlicher den Aufbau des Programmstücks, in das der Makroaufruf expandiert wird; das Ergebnis der Expansion ist hier die Anwendung eines LAMBDA -Ausdrucks.

Die Nützlichkeit von "Backquote" zeigt sich auch bei einer Neudefinition von OUR-IF.
Wir hatten OUR-IF bisher wie folgt definiert:

```
(DEFMACRO OUR-IF (A B &OPTIONAL C)
   (LIST   'COND
                (LIST A B)
                (LIST 'T C)))
```

Mit Backquote ist folgende Definition möglich :

```
(DEFMACRO OUR-IF (A B &OPTIONAL C)
   `(COND (,A ,B)
          ( T ,C)))
```

Die zweite Definition zeigt im Rumpf viel deutlicher das Muster des Programmstücks, in das der Makroaufruf expandiert wird.

15.5 Seiteneffekte bei Makros

In diesem Kapitel wollen wir weitere Beispiele für Anwendungen von Makros kennenlernen und dabei besonders auf die Möglichkeit von Seiteneffekten achten.
Wir betrachten eine Realisierung der "Kelleroperation" Push zum Einfügen eines Elements in einen sogenannten Keller.
Wenn wir den Keller durch eine Liste darstellen, dann können wir Push-Operationen durch "AnCONSen" von Elementen beschreiben.
Wir betrachten zunächst die Definition einer Funktion PUSH und ihre Wirkung und geben anschließend zum Vergleich eine Definition als Makro:

```
-->(DEFUN PUSH (EL STACK)
     (SETQ STACK (CONS EL STACK)))
PUSH

-->(SETQ ELEMENT 'A KELLER '(B C D))
(B C D)

-->(PUSH ELEMENT KELLER)                ; Aufruf von PUSH
(A B C D)

-->KELLER
(B C D)                                 ; der Keller wurde nicht verändert
```

Jetzt definieren wir PUSH als Makro und betrachten den entsprechenden Aufruf

```
-->(DEFMACRO PUSH (EL STACK)
            `(SETQ ,STACK
                   (CONS ,EL ,STACK)))
PUSH

-->(PUSH ELEMENT KELLER)
(A B C D)

-->KELLER
(A B C D)
```

Durch den Makroaufruf wurde der Keller verändert ! Um die Ursache zu erkennen, betrachten wir die Makroexpansion des obigen Makroaufrufs:

Bei der Makroexpansion wird der Makrorumpf ausgewertet und zwar unter Bindung der Parameter an die unausgewerteten Argumente.

```
-->(MACROEXPAND '(PUSH ELEMENT KELLER))
(SETQ KELLER (CONS ELEMENT KELLER))

T
```

Die Makroexpansion liefert also

```
(SETQ KELLER (CONS ELEMENT KELLER))
```

als auszuwertendes Programmstück. Bei seiner Auswertung wird offensichtlich das Argument KELLER verändert.

15.6 Zusammenfassung

Makros haben bemerkenswerte Eigenschaften:

- Makros ermöglichen das Erstellen von Programmen, die selbst Programme erstellen.
  Dies ist eine Konsequenz aus der Tatsache, daß LISP-Programme und LISP-Daten dieselbe Gestalt haben.

- Makros übernehmen ihre Argumente unausgewertet;
  sie ermöglichen damit auch dem Benutzer die Definition von speziellen Formen und - wie am Beispiel von IF und LET gezeigt wurde - von Kontrollstrukturen.

- Makros erledigen ihre Arbeit in zwei Schritten:
  Zuerst wird mit unausgewerteten Argumenten durch Makroexpansion eine Zwischenform aufgebaut, die anschließend ausgewertet wird. Das Resultat dieser zweiten Auswertung gibt das Resultat der Makroauswertung.
  Bei Verwendung des LISP-Interpreters werden beide Schritte hintereinander von diesem ausgeführt. Bei Verwendung eines Compilers wird die Makroexpansion von diesem vorgenommen und das übersetzte Programm zur Laufzeit ausgewertet. Der Unterschied in der Effizienz ist enorm. Das Makrokonzept zeigt seine Vorzüge insbesondere bei der Übersetzung.

- Makros eignen sich insbesondere auch, um komplexere Datenstrukturen einschließlich der zugehörigen Zugriffsfunktionen (entsprechend dem Konzept der "abstrakten Datentypen") zu realisieren. Ein einfaches Beispiel gab das Makro PUSH.

Makros werden definiert mit Hilfe des Makros DEFMACRO. Der Rumpf eines Makros gibt die Expansionsfunktion. "Backquote" und "Comma" geben die Möglichkeit einer Quotierung, die lokal rückgängig gemacht werden kann. Damit wird das Schreiben von Makros wesentlich erleichtert.

KAPITEL 16:

Symbole

Symbole wurden bisher ausschließlich als Bezeichnungen oder Namen für Variable und Funktionen verwendet.

Tatsächlich sind Symbole in COMMON LISP kompliziertere Datenobjekte, die verschiedenen Zwecken dienen können. Mit diesen Möglichkeiten beschäftigt sich das vorliegende Kapitel.

Als erstes gehen wir nochmals auf Namen für Symbole ein. Dann behandeln wir den Zugriff auf Symbole. Dabei zeigt sich, daß die bisherige Form des Zugriffs auf ein Symbol über "seinen" Namen in einem allgemeineren Zusammenhang gesehen werden muß. Die nachfolgenden Kapitel beschäftigen sich mit der Erzeugung von Symbolen, der "Struktur" von Symbolen, der sogenannten Property-Liste von Symbolen und schließlich dem Zugriff auf die einem Symbol zugeordneten Werte und Funktionen.

16.1 Namen für Symbole

Wir haben Symbole bisher in der Gestalt von Zeichenfolgen geschrieben und dabei praktisch alle Zeichenfolgen zugelassen, die von Zahlen, Satzzeichen und Klammern unterscheidbar und nicht mit Anführungszeichen eingerahmt sind. In Ergänzung dazu legen wir jetzt noch fest, daß die Zeichen [,], (,), ! und ? in Symbolen normalerweise nicht auftreten.
Weiter weisen wir darauf hin, daß zum Schreiben von Symbolen sowohl Großbuchstaben wie Kleinbuchstaben verwendet werden dürfen. Allerdings werden Kleinbuchstaben beim Einlesen normalerweise in Großbuchstaben umgewandelt.
Demnach bezeichnen die Zeichenfolgen
FROBBOZ,
frobboz
fRObBoz
grundsätzlich dasselbe Symbol.
Weitere Beispiele für Symbole sind:

+$
CASH-AND-CARRY
1+
PASCAL-STYLE
/olympic/games/LA
file.rel.43
Dagegen ist +1 kein Symbol.

Wenn ein Symbol nicht einfach mit den Zeichen seines Namens niedergeschrieben werden kann, weil der Name entweder nicht zugelassene Zeichen oder definitiv Kleinbuchstaben enthalten soll, dann können die "Fluchtsymbole" \ und | weiterhelfen.
Jedes Erscheinen des Zeichens \ vor einem beliebigen Zeichen bewirkt, daß das zweite Zeichen wie ein normales Zeichen in einem Symbolnamen behandelt wird; insbesondere wird die interne Umwandlung von Kleinbuchstaben in Großbuchstaben unterdrückt.
Weiter kann ein Zeichen, dem ein \ voransteht, niemals als Zahl interpretiert werden.

Beispiele für Namen von Symbolen sind damit:
\(
\+1
+\1
\froboz
3.14159\s1
APL\\360
apl\\360
\(b↑2\)\ -\ 4*\a*\ b

Die Mühe ein \ eventuell mehrfach in einem Namen schreiben zu müssen, kann man sich sparen, wenn man den ganzen Namen mit senkrechten Strichen einrahmt. Dies hat dieselbe Wirkung wie ein \ vor jedem einzelnen Zeichen.
Hierzu gibt es zwei Ausnahmen:
Den Fluchtsymbolen | und \ selbst muß auf jeden Fall ein \ voranstehen.

Beispiele:
| " |
|(b ↑ 2) - 4*a*c|
|froboz|
|APL \\ 360|
|\|\| |

Ob ein LISP-Objekt ein Symbol darstellt, kann mit Hilfe des Prädikats SYMBOLP festgestellt werden. Es hat das **Aufrufmuster :**

(SYMBOLP <Objekt>)

Wert:

T	falls der Wert des Arguments ein Symbol ist
NIL	sonst

Der Name eines Symbols wird häufig auch als sein Print-Name (abgekürzt P-Name) bezeichnet, weil er insbesondere zur externen Repräsentation benutzt wird. Bisher haben wir Symbole stets mit diesem ihren PRINT-Namen identifiziert.

16.2 Zugriff auf Symbole über Packages

Die in einem System jeweils verwendeten Symbole müssen natürlich verwaltet werden. In früheren LISP-Systemen wurden alle Symbole, die im System gerade verwendet wurden, in einer sogenannten ObList (für **Object List**) geführt und über ihren Namen identifiziert. In COMMON LISP werden die im System jeweils verwendeten Symbole auf sogenannte Packages verteilt.
Der Zugriff auf ein Symbol erfordert deshalb neben der Angabe des Symbolnamens auch die (explizite oder implizite) Angabe eines Package, dem das Symbol angehört.

Dabei können die Namen verschiedener Symbole durchaus übereinstimmen, wenn sie verschiedenen Packages angehören. Nur innerhalb eines Package müssen die Symbolnamen eine eindeutige Identifikation ermöglichen, also verschieden sein.

In großen Programmsystemen mit Modulen, die von vielen verschiedenen Programmierern geschrieben werden, bilden nämlich zufällige Namenskonflikte ein ernstes Problem. Dies wird mit dem Package-Konzept von COMMON LISP vermieden. Zugleich wird die modulare Struktur großer Programm-Systeme durchsichtiger.
Bei der Abarbeitung eines Programms ist zu jedem Zeitpunkt genau ein Modul und damit ein Package aktuell. Dieses ist jeweils durch den Wert der Variablen *PACKAGE* gegeben und wird vom LISP-System für den Zugriff auf ein Symbol zugrundegelegt, wenn es den Namen eines Symbols ohne zusätzliche Angaben liest.

Der Zugriff auf ein Symbol kann aber auch über ein anderes als das aktuelle Package erfolgen;

dabei ist aber mit dem Namen des Symbols auch dasjenige Package anzugeben, über das der Zugriff erfolgen soll. Auf Details der zugehörigen Notation gehen wir nicht ein.

Innerhalb eines (aktuellen) Package kann jedoch i.a. nicht nur auf diejenigen Symbole zugegriffen werden, die dem Package unmittelbar angehören. Ein Zugriff auf Symbole aus anderen Packages kann auf verschiedene Weisen ermöglicht werden: Die Symbole, auf die man in einem Package zugreifen kann, werden unterschieden in interne und externe Symbole. Interne Symbole eines Package sind aus anderen Packages nicht zugänglich. Externe Symbole dienen der Kommunikation mit anderen Packages. Die meisten Symbole werden als interne Symbole erzeugt; sie werden extern, wenn sie explizit in einem sogenannten EXPORT-Befehl auftreten. Der Zugriff auf externe Symbole aber kann auch anderen Packages eröffnet werden.

Ein einzelnes externes Symbol eines Package kann in einem anderen Package durch einen IMPORT-Befehl zugänglich gemacht werden. Es wird dann dort internes Symbol. Diese Importierung kann mit einem UNINTERN-Befehl auch wieder rückgängig gemacht werden.
Alle externen Symbole eines Package können in einem anderen Package durch einen USE-PACKAGE-Befehl zugänglich gemacht werden. Sie werden dann alle interne Symbole des anderen Package. Man spricht hier von "Vererbung"; auch diese kann mit einem UNUSE-PACKAGE-Befehl wieder rückgängig gemacht werden.

In beiden Fällen ist dann der Zugriff auch aus dem anderen Package möglich und zwar ohne Angabe eines Package-Namens, allein durch Angabe des Symbolnamens, also des P-Namens.
Diese Eröffnung von Zugriffsmöglichkeiten kann natürlich Namenskonflikte mit sich bringen. Diese werden vom LISP-System bei jeder Anwendung der Funktionen IMPORT und USE-PACKAGE ermittelt und dem Benutzer mitgeteilt; er muß dann durch sogenannte Shadow-Befehle festlegen, welche Symbole Priorität vor anderen Symbolen gleichen Namens erhalten sollen, die ansonsten auch zugänglich wären. Auf Details der Manipulation von Packages gehen wir hier nicht ein. Wir erwähnen aber noch einige Standard-Packages.

Jedes LISP-System verfügt standardmäßig zumindest über Packages mit den Namen LISP, USER, KEYWORD und SYSTEM.

Das Package USER wird bei jedem Start des COMMON-LISP-Systems automatisch das aktuelle Package. Es enthält alle vom Benutzer in der bisher beschriebenen Weise eingeführten Symbole und benutzt darüberhinaus das Package LISP.

Das Package LISP enthält alle Standardfunktionen des LISP-Systems. Seine externen Symbole umfassen alle dem Benutzer zugänglichen Funktionen und alle globalen Variablen wie CAR,

CDR , *PACKAGE* usw. Fast alle anderen Packages nutzen LISP, so daß diese Symbole ohne Angabe eines Package zugänglich sind.

Das Package KEYWORD enthält alle Schlüsselwörter, die in vom LISP-System bereitgestellten oder vom Benutzer definierten Funktionen auftreten. Jede Symbolbezeichnung, die mit einem Doppelpunkt beginnt, wird als Bezeichnung eines Symbols in diesem Package interpretiert.

Das Package SYSTEM ist für die Implementation reserviert. Es benutzt LISP und hat den Spitznamen SYS. Für Packages können nämlich neben der eigentlichen Bezeichnung, die i.a. die Bedeutung dokumentiert, auch beliebig viele Spitznamen (nicknames) eingeführt und als Kurzbezeichnungen verwendet werden.

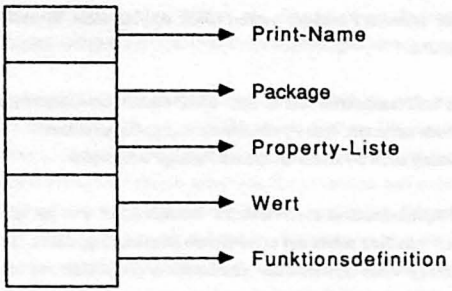
16.3 Zur Erzeugung von Symbolen

Bisher haben wir Symbole dadurch erzeugt, daß wir sie in einem LISP-Programm verwendet haben. Immer, wenn das COMMON-LISP System einen P-Namen liest, sucht es im Package-System nach dem zugehörigen Symbol. Wenn diese Suche erfolglos ist, wird automatisch ein entsprechendes Symbol angelegt; es wird dabei Eigentum des jeweils aktuellen Package. Dies ist der Normalfall. Die entsprechenden Symbole heißen "interned symbols".

Man kann aber auch Symbole erzeugen, die keinem Package angehören; sie heißen "uninterned symbols" und müssen vor ihrer ersten Verwendung durch eigene Erzeugungsfunktionen erzeugt werden. Diese Erzeugung ist vor allem dann von Bedeutung, wenn es darauf ankommt, Symbole zu erzeugen, deren P-Name mit Sicherheit im bisherigen Programm noch nicht aufgetreten sind. Eine der bekanntesten Erzeugungsfunktionen ist GENSYM von generate symbol. Wir gehen hierauf an dieser Stelle nicht näher ein.

16.4 Zur Struktur von Symbolen

Ein LISP-Symbol ist ein strukturiertes Datenobjekt; für den Benutzter sind folgende Komponenten interessant und sichtbar:



Die Komponenten eines Symbols

Der **Print-Name** (abgekürzt P-Name) dient vor allem der externen Repräsentation. Er ist gegeben durch eine Zeichenfolge.

Die **Package-Zelle** enthält einen Zeiger auf ein Package. Wie wir gesehen haben, ist ein Symbol durch seinen Namen nur bezüglich eines einzigen Package eindeutig identifizierbar. Ein Symbol kann in vielen Packages zugänglich sein, es gehört ("is owned by") aber stets nur höchstens einem Package, seinem "Home-Package" oder "Eigentümer" ("owner") an. Die Package-Zelle enthält einen Zeiger auf das Home-Package, falls ein solches existiert.

Die **Property-Liste** ist eine Liste von Eigenschafts-Werte- Paaren, die jedem Symbol zugeordnet sein können. Wir gehen darauf im nächsten Kapitel ein.

Diese drei Komponenten sind immer vorhanden.

Ob die **Wert-Zelle** und die **Funktions-Zelle** vorhanden sind, hängt von der Implementierung ab. Falls vorhanden, enthält die Wert-Zelle einen Zeiger (Pointer) auf denjenigen Speicherbereich, der den aktuellen Wert des Symbols enthält.

Falls die Funktions-Zelle vorhanden ist, enthält sie einen Zeiger (Pointer) auf einen Speicherbereich, in dem eine dem P-Namen zugeordnete Funktionsdefinition abgelegt ist.

16.5 Die Property-Liste

In der Sprache LISP wird immer schon jedem Symbol in Form der sogenannten Property-Liste (kurz P-Liste, engl. plist) eine Tabelle von Eigenschafts-Werte-Paaren zugeordnet. Eine derartige Property-Liste enthält null oder mehr Einträge. Jeder Eintrag besteht aus einem Symbol zur Bezeichnung einer Eigenschaft (dem Schlüssel oder Indikator) und einem Wert für diese Eigenschaft.

Ähnliches kennen wir von Assoziationslisten, doch sind folgende Unterschiede wesentlich:
Assoziations-Listen sind Listen, deren Elemente selbst zweielementige Listen sind.
Property-Listen sind Listen mit einer geraden Anzahl von Elementen, die abwechselnd die Indikatoren und die zugehörigen Werte bilden. Anders als bei Assoziationslisten sind jedoch Wiederholungen unter den Indikatoren nicht zugelassen und alle Operationen für Hinzufügen oder Entfernen von Eigenschafts-Werte-Paaren sind destruktiv.

Wenn ein Symbol erzeugt wird, ist die zugehörige Property-Liste zunächst leer.

Aus der P-Liste eines Symbols kann mit der Funktion GET gelesen werden und zwar mit dem **Aufrufmuster :**

(GET <-Symbol-> <-Indikator-> <-Default-Wert->)

> Die ersten beiden Argumente sind obligatorisch; ihre Werte bestimmen ein Symbol und einen Indikator für eine Eigenschaft. Das dritte Argument ist optional; sein Wert dient als Default-Wert.

Auswertung:
Die dem Symbol zugeordnete P-Liste wird nach einem Indikator durchsucht, der mit dem Wert des zweiten Arguments bezüglich EQ übereinstimmt. Enthält die P-Liste einen solchen Indikator, so wird der zugehörige Eigenschaftswert übergeben; ist ein solcher Indikator nicht vorhanden, so wird der Wert des optionalen Arguments als Default-Wert übergeben. Fehlt das optionale Argument, so dient NIL als Default-Wert. Ob ein Indikator nicht vorhanden oder vorhanden ist und als Eigenschaftswert den Default-Wert NIL hat, kann dabei nicht unterschieden werden.
Seiteneffekt: Keiner

Beispiel:
Das Symbol FOO habe die Property-Liste (BAR T RICH VERY DOG "GREY")

Dann gilt:

-->(GET 'FOO 'RICH)
VERY

-->(GET 'FOO 'ZOO)
NIL

Ein neues Eigenschafts-Werte-Paar in der P-Liste eines Symbols kann mit Hilfe des allgemeinen Zuweisungsmakros SETF gebildet werden:
Ein Aufruf der **Gestalt**

(SETF (GET <-Symbol-> <-Indikator->) <-Wert->)

bewirkt als **Seiteneffekt,**
daß dasjenige Datenobjekt, das ansonsten von der GET-Form geliefert würde, durch den Wert des zweiten Arguments ersetzt wird. Falls die P-Liste des Symbols den gegebenen Indikator enthält, wird der zugehörige Eigenschaftswert verändert, anderenfalls wird die P-Liste um einen entsprechenden Indikator-Wert-Eintrag erweitert.

Wert der SETF-Form wird der Wert ihres zweiten Arguments.

Beispiel:
Das Symbol FOO habe wie oben die P-Liste (BAR T RICH VERY DOG "GREY")

Dann gilt:

-->(SETF (GET 'FOO 'ZOO) 'ELEFANTS)
ELEFANTS

-->(GET 'FOO 'ZOO)
ELEFANTS

Die Entfernung eines Eigenschafts-Werte-Paares aus der P-Liste eines Symbols ist möglich mit Hilfe der Funktion REMPROP ; sie hat das **Aufrufmuster :**

(REMPROP <-Symbol-> <-Indikator->)

Auswertung:
In der P-Liste des (als Wert des ersten Arguments gegebenen) Symbols wird ein Indikator gesucht, der bezüglich EQ mit dem Wert des zweiten Arguments übereinstimmt.
Wird ein solcher Indikator gefunden, so bewirkt der **Seiteneffekt** von REMPROP die Entfernung dieses Indikators und des zugehörigen Eigenschaftswerts aus der P-Liste; sodann wird ein implementationsabhängiger Nicht-NIL-Wert als Wert der Form übergeben. Wird kein entsprechender Indikator gefunden, so wird die P-Liste nicht verändert, und NIL wird als Wert der Form übergeben.

Beispiel:
Das Symbol FOO habe die Property-Liste (CAR BLACK HEIGHT 2.9 CAT WHITE)
Dann bewirkt die Auswertung von

(REMPROP 'FOO 'HEIGHT),

daß die Property-Liste abgeändert wird in

(CAR BLACK CAT WHITE)

und ein Nicht-NIL-Wert übergeben wird.

Die gesamte P-Liste eines Symbols als alternierende Liste von Eigenschaftsnamen und Werten kann durch Anwenden der Funktion SYMBOL-PLIST ermittelt werden.

Aufrufmuster:

(SYBMOL-PLIST <-Symbol->)

Wert:
Die gesamte P-Liste des Symbols.

Seiteneffekt: Keiner

Mit

(SETF (SYMBOL-PLIST <-Symbol->) . . .)

kann nun die ganze P-Liste eines Symbols destruktiv ersetzt werden. Diese Operation ist allerdings sehr gefährlich, da sie auch Informationen zerstört, die das System möglicherweise selbst in Property-Listen angelegt hat.

Die Funktionen **GET** und **REMPROP** benötigen als Wert des ersten Arguments ein Symbol und wirken auf dessen P-Liste. Sie sind nicht anwendbar auf beliebig gegebene Listen im P-Listen-Format, das sind Listen mit gerader Anzahl von Elementen, die abwechselnd Symbole und zugehörige Eigenschaftswerte darstellen. GET und REMPROP sind deshalb auch nicht anwendbar auf das Ergebnis der Anwendung von SYMBOL-PLIST.
Als Ersatz dienen hier die Funktionen **GETF** und **REMF**. Sie wirken auf beliebig gegebene Listen im P-Listen-Format genauso wie GET und REMPROP auf die P-Liste eines Symbols.
Sie haben die **Aufrufmuster :**

(GETF <Platzangabe> <-Indikator-> <-Default-Wert>)

(REMF <Platzangabe> <-Indikator->)

Das erste Argument gibt jeweils an, wo die Liste zu finden ist, die manipuliert werden soll. Ist der Wert des ersten Arguments ein Symbol, dann wird dessen P-Liste manipuliert. Ist der Wert eine beliebige Liste im Format einer P-Liste, dann wird diese manipuliert.

Beispiele:

```
-->(GETF '(BAR T RICH VERY) RICH)
VERY

-->(REMF '(CAR BLACK CAT WHITE) 'CAT)
(CAR BLACK)
```

16.6 Symbole für Werte und Funktionsbeschreibungen

Wir kommen nochmals auf die ursprüngliche Bedeutung von Symbolen als Träger von zugeordneten Werten und Funktionsbeschreibungen zurück. Ob für ein Symbol solche Zuordnungen bestehen oder nicht, kann mit entsprechenden Prädikaten festgestellt werden. Diese

Prädikate und Zugriffsfunktionen wollen wir jetzt kennenlernen. Wir benötigen sie auch bei der Konstruktion eines kleinen LISP-Interpreters in der nächten Lektion.

Ob die von einem Symbol benannte globale Variable einen Wert hat oder nicht, testen wir mit dem Prädikat BOUNDP. Den gegebenenfalls vorhandenen Wert bestimmen wir mit der Funktion SYMBOL-VALUE.

Aufrufmuster von BOUNDP:
(BOUNDP <Symbol>)

Wert:

T	wenn die vom gegebenen Symbol benannte globale Variable einen Wert hat
NIL	sonst.

Seiteneffekt: Keiner

Aufrufmuster von SYMBOL-VALUE:

(SYMBOL-VALUE <Symbol>)

Wert:

Wert des Symbols,	wenn dem Symbol ein Wert zugeordnet ist,
Fehlermeldung	sonst.

Seiteneffekt: Keiner

Ob einem Symbol global eine Funktion zugeordnet ist, testen wir mit dem Prädikat FBOUNDP; die gegebenenfalls zugeordnete Funktion ermitteln wir mit der Funktion SYMBOL-FUNCTION.

FBOUNDP hat das **Aufrufmuster :**

(FBOUNDP <Symbol>)

Wert:

T, wenn dem Symbol global eine Funktion zugeordnet ist; dies schließt auch den Fall ein, daß dem Symbol eine spezielle Funktion oder ein Makro zugeordnet ist.

Nil sonst.

Seiteneffekt: Keiner

SYMBOL-FUNCTION hat das **Aufrufmuster :**

(SYMBOL-FUNCTION <Symbol>)

Wert:

Die dem Symbol global zugeordnete Funktionsdefinition, falls eine solche Zuordnung besteht
Fehlermeldung sonst.

Seiteneffekt: Keiner

Für ein beliebiges LISP-Objekt kann festgestellt werden, ob es ein funktionales Objekt ist; dazu dient das Prädikat FUNCTIONP mit dem

Aufrufmuster :
(FUNCTIONP <Objekt>)

Wert:

T, falls das Objekt ein funktionales Objekt ist, das auf Argumente angewendet werden kann, zum Beispiel mit Hilfe von FUNCALL oder APPLY.

NIL sonst.

Seiteneffekt: Keiner

FUNCTIONP liefert insbesondere den Wert T für Lambda-Ausdrücke, jeden von der speziellen Form FUNCTION übergebenen Wert sowie jede durch den Compiler übersetzte Funktion.

16.7 Zusammenfassung

Symbole sind Atome, genauso wie Zahlen oder Strings, haben aber eine komplizierte Struktur mit verschiedenen Komponenten. Durch diese wird u.a. festgelegt

- der Print-Name des Symbols; er dient der externen Darstellung und der Identifikation innerhalb seines Package
- das Home-Package des Symbols; es "besitzt" (englisch owns) das Symbol; in ihm wird es durch seinen Namen eindeutig identifiziert
- die Property-Liste; sie enthält modifizierbare Komponenten des Symbols
- der Wert des Symbols
- die dem Symbol gegebenenfalls zugeordnete Funktion.

Das Package-Konzept unterstützt den modularen Aufbau von Programmsystemen. Sofern der Benutzer nichts anderes explizit festlegt, ist das Package USER aktuell. Es "besitzt" (= owns) alle vom Benutzer in der bisher beschriebenen Weise eingeführten Symbole und hat Zugriff auf das Package LISP, dem alle Symbole für die vom System bereitgestellten "primitiven" Objekte angehören. Für Zugriff auf das Package KEYWORD, das alle vom System oder Benutzer definierten Schlüsselwörter enthält, wird die Package-Kennung in Gestalt eines Doppelpunktes vor dem Symbol angegeben.

KAPITEL 17:

Ein LISP-Interpreter

In diesem Kapitel soll ein Interpreter für eine Teilmenge von COMMON-LISP entwickelt werden. Dazu definieren wir eine Funktion O-EVAL (für OUR-EVAL), die ihrerseits eine Vielzahl von Hilfsfunktionen aufruft. Die Funktion O-EVAL und die zugehörigen Hilfsfunktionen werden im folgenden nach dem Top-down-Prinzip entwickelt, d.h. ausgehend vom allgemeinen Ziel wird durch sukzessive Modularisierung ein vollständiges (hierarchisch gegliedertes) Programm erarbeitet.

17.1 Ein Interpreter für LISP in LISP

Ein LISP-Interpreter ist ein Programm (für die Funktion EVAL), das in einer beliebigen Programmiersprache geschrieben sein kann, z.B. in einer Assembler-Sprache.

Wir beschreiben im folgenden einen LISP-Interpreter und zwar in LISP selbst ! Die Frage,welchen Sinn dies hat, ist zunächst berechtigt. Denn ein LISP-Interpreter, geschrieben in LISP, ist ein LISP-Programm, das nur mit einem anderen LISP-Interpreter oder einem LISP-Compiler abgearbeitet werden kann.

Bei Abarbeitung mit einem anderen Interpreter wird dieser benutzt, um den ersten Interpreter zu interpretieren. Das ist zum Beispiel dann sinnvoll, wenn ein LISP-Interpreter für einen speziellen LISP-Dialekt vorliegt, ein Interpreter für einen anderen LISP-Dialekt schnell entwickelt werden soll, und zunächst die Effizienz der Programmverarbeitung weniger wichtig ist. Bei Vorliegen eines Compilers kann der in LISP geschriebene Interpreter mit dem LISP-Compiler in die Zielsprache des Compilers, sagen wir die Maschinensprache einer Maschine M, übersetzt werden. Der übersetzte Interpreter kann dann als LISP-Interpreter auf der Maschine M eingesetzt werden.

Über diese möglichen Anwendungen hinaus ist der Entwurf eines LISP-Interpreters in LISP ganz besonders geeignet, unser Verständnis für die Sprache LISP zu fördern.

Um die Konstruktion des LISP-Interpreters übersichtlich gestalten zu können, beschränken wir ihn auf eine noch zu definierende Teilmenge (englisch subset) der Sprache COMMON LISP. Diese Teilmenge ist charakterisiert durch die Beschränkung auf

- die Systemfunktionen ATOM, CAR, CDR, CONS, EQ sowie ZEROP und + , wir nennen sie die primitiven Operationen ,
- die speziellen Formen IF und QUOTE ,
- die vom Benutzer definierten benannten Funktionen und Lambda-Ausdrücke.

Alle unter dieser Beschränkung definierbaren Ausdrücke müssen von unserem LISP-Interpreter verarbeitet werden können.

17.2 Die Funktion O-EVAL beschreibt den Interpreter

Die Funktion O-EVAL soll eine vorgegebene Form in der jeweils aktuellen Umgebung auswerten können. Dementsprechend enthält die Definition von O-EVAL die Parameter FORM und ENV. An FORM wird als obligatorisches Argument der auszuwertende Ausdruck übergeben; an ENV wird als optionales Argument eine Assoziationsliste (kurz A-Liste) übergeben, die alle lokalen Bindungen enthält und zusammen mit der globalen Umgebung die aktuelle Umgebung für die Auswertung der Form festlegt. Bei Fehlen eines Arguments für ENV wird der Default-Wert NIL - also eine leere A-Liste - zugrunde gelegt. Die korrekte Bindung der aktuellen Umgebung erhält man für jedes Symbol, indem man zuerst in der A-Liste sucht und erst nach erfolglosem Durchsuchen derselben die Zuordnung aus der globalen Umgebung entnimmt.

Die Funktion O-EVAL muß bei Auswertung einer Form folgende Fälle berücksichtigen:

Die auszuwertende Form ist

entweder	ein Symbol: dessen Wert soll mit einer noch zu definierenden Funktion O-EVAL-SYMBOL ermittelt werden,
oder	eine nichtleere Liste (daran erkennbar, daß sie CONSP erfüllt), die den Aufruf einer Funktion beschreibt: sie soll mit einer noch zu definierenden Funktion O-EVAL-CONS ausgewertet werden,
oder	als einzig verbleibende Möglichkeit ein Atom, das kein Symbol, sondern eine Zahl, ein String oder einer der Werte T und NIL ist: der Wert dieses Atoms wird unmittelbar übergeben.

Damit erhalten wir folgende Definition für O-EVAL:

```
(DEFUN O-EVAL (FORM &OPTIONAL (ENV 'NIL))
  (COND     ((SYMBOLP FORM) (O-EVAL-SYMBOL FORM ENV))
            ((CONSP FORM) (O-EVAL-CONS FORM ENV))
            (T FORM)))
```

Die Hilfsfunktionen O-EVAL-SYMBOL und O-EVAL-CONS sind jetzt noch zu definieren. Jede von ihnen benötigt zwei obligatorische Parameter, die im angegebenen Aufruf an die Werte von FORM und ENV gebunden werden. Der Einfachheit halber bezeichnen wir auch diese Parameter selbst mit FORM und ENV.

Ihre Definitionen beginnen demnach mit

```
(DEFUN O-EVAL-SYMBOL (FORM ENV) ...
```

bzw. mit

```
(DEFUN O-EVAL-CONS (FORM ENV) ...
```

und werden in den nächsten Unterkapiteln erarbeitet.

17.2.1 Zur Auswertung von Symbolen

Die Funktion O-EVAL-SYMBOL wird aufgerufen, wenn FORM ein Symbol darstellt, und soll dessen Wert ermitteln. Dabei ist zuerst zu prüfen, ob das Symbol eine lokale Variable ist, hierzu ist die A-Liste für ENV nach einem entsprechenden Eintrag zu durchsuchen, oder ob das Symbol eine globale Variable ist. Liegt keiner dieser beiden Fällen vor, so ist eine Fehlermeldung auszugeben. Dies führt uns auf folgende Definition

```
(DEFUN O-EVAL-SYMBOL (FORM ENV)
  (LET ((ENTRY (ASSOC FORM ENV)))
       (COND  ((NOT (NULL ENTRY)) (CDR ENTRY))
              ((BOUNDP FORM) (SYMBOL-VALUE FORM))
              (T (ERROR "Unbound Variable: ~S" FORM)))))
```

Zuerst wird geprüft, ob das durch FORM dargestellte Symbol eine lokale Variable ist. Dazu wird mit ASSOC in der durch ENV gegebenen A-Liste nach einem entsprechenden Eintrag gesucht.

Wird ein nichtleerer Eintrag gefunden, so ergibt seine zweite Komponente (CDR ENTRY) den Wert des Symbols. Wird kein Eintrag gefunden, so wird mit BOUNDP geprüft, ob die globale Umgebung eine Wertzuordnung für das Symbol enthält; wenn ja, wird der Wert mit SYMBOL-VALUE ermittelt. Wird auch in der globalen Umgebung keine Wertzuordnung festgestellt, so wird eine Fehlermeldung ausgegeben.

17.2.2 Zur Auswertung von nichtleeren Listen

Zur Auswertung von nichtleeren Listen verwenden wir die Funktion O-EVAL-CONS. Wie wir sahen, wird sie im Rahmen der Abarbeitung eines Aufrufs von O-EVAL genau dann mit

(O-EVAL-CONS FORM ENV)

aufgerufen, wenn FORM das Prädikat CONSP erfüllt. In diesem Fall stellt FORM eine Liste dar, die den Aufruf einer Funktion beschreibt, der in der aktuellen Umgebung auszuwerten ist. Die zugehörige Funktionsbeschreibung ist gegeben durch (CAR FORM), die Liste der Argumente durch (CDR FORM).

Die Definition von O-EVAL-CONS gewinnt damit folgende Gestalt:

```
(DEFUN O-EVAL-CONS (FORM ENV)
   (LET ((FN (CAR FORM))
         (ARGS (CDR FORM)))
         <Rumpf der LET-Form>))
```

In der LET-Form mit den lokalen Variablen FN und ARGS für Funktionsbeschreibung und Argumente wird die Funktion ausgewertet. Das Vorgehen bei der Auswertung ist natürlich von der Art der Funktion abhängig. Deshalb ist der Rumpf der LET-Form eine COND-Form. Die Klauseln entsprechen den verschiedenen Möglichkeiten für die Art der Funktion:

```
(COND   (<Spezielle Form ? >          <zugehörige Auswertung>)
        (<Primitive Operation ? >     <zugehörige Auswertung>)
        (<Lambda-Ausdruck ? >         <zugehörige Auswertung>))
        (<Frei definierte Funktion ? > <zugehörige Auswertung>)
```

Die verschiedenen Fälle werden im folgenden behandelt.

17.2.2.1 Spezielle Formen und ihre Auswertung

Als erstes müssen wir testen, ob eine spezielle Form vorliegt. Das Testprädikat wird aufgerufen mit (SP-FORM-P FN); es testet, ob FN mit dem Wert (CAR FORM) eine der in unserem LISP-Subset zugelassenen speziellen Formen IF oder QUOTE ist.
Wir definieren es mit

```
(DEFUN SP-FORM-P (FN)
   (MEMBER FN '(QUOTE IF)))
```

Liegt eine spezielle Form vor, so werten wir diese aus mit dem Funktionsaufruf

```
(O-EVAL-SP-FORM FN ARGS ENV)
```

wobei folgende Definition zugrundeliegt:

```
(DEFUN O-EVAL-SP-FORM (FN ARGS ENV)
   (COND      ((EQ FN 'QUOTE) (CAR ARGS))
              ((EQ FN 'IF)
               (COND ((O-EVAL   (CAR ARGS) ENV)
                                (O-EVAL (CADR ARGS) ENV))
                     (T (O-EVAL (CADDR ARGS) ENV))))))
```

Wenn der Wert von (CAR FORM), also der Wert von FN, gleich dem Symbol QUOTE ist, ergibt der Wert (CDR FORM), also der Wert von ARGS, nach Entfernen des äußeren Klammerpaares den Wert der Form.

Wenn der Wert von FN gleich dem Symbol IF ist, wird in Abhängigkeit vom Wert von (CAR ARGS) entweder (CADR ARGS) oder (CADDR ARGS) ausgewertet und als Wert der Form übergeben.

So also werden spezielle Formen ausgewertet.

17.2.2.2 Primitive Operationen und ihre Auswertung

Als nächstes betrachten wir die Auswertung von primitiven Operationen.
Zuerst testen wir mit (PRIM-OP-P FN), ob FN eine der Operationen ATOM , CAR , CDR , CONS , EQ , + oder ZEROP darstellt.
Das Testprädikat PRIM-OP-P definieren wir wie folgt:

```
(DEFUN PRIM-OP-P (FN)
    (MEMBER FN '(ATOM CAR CDR CONS EQ + ZEROP)))
```

Zeigt dieser Test, daß FN eine primitive Operation darstellt, so wird diese ausgewertet mit der Funktion O-EVAL-PRIM-OP, die ihrerseits zwei Argumente, nämlich FN und die Liste der Argumentwerte für FN erwartet. Diese Liste der Argumentwerte wird ermittelt durch Auswertung von

```
(O-EVAL-LIST ARGS ENV)
```

Die Funktionen O-EVAL-LIST und O-EVAL-PRIM-OP werden anschließend definiert.

Die Auswertung der durch FN gegebenen primitiven Operation für die durch die Liste ARGS gegebenen Argumente und die durch die A-Liste ENV gegebene Umgebung kann demnach mit dem Funktionsaufruf

```
(O-EVAL-PRIM-OP FN (O-EVAL-LIST ARGS ENV))
```

vorgenommen werden.

Wir kommen nun zur Definition von O-EVAL-LIST:

```
(DEFUN O-EVAL-LIST (FORMS ENV)
   (COND     ((NULL FORMS) NIL)
             (T (CONS   (O-EVAL (CAR FORMS) ENV)
                        (O-EVAL-LIST (CDR FORMS) ENV)))))
```

Die Funktion O-EVAL-LIST übernimmt über den Parameter FORMS eine Liste von Formen. Sie wertet diese Formen der Reihe nach aus und zwar unter Zugrundelegung der aktuellen Umgebung.

welche durch ENV und die globale Umgebung bestimmt ist. Die Werte der Listenelemente werden wieder zu einer Liste zusammengefaßt, die dann als Wert übergeben wird.

Jetzt zur Funktion **O-EVAL-PRIM-OP:**
Sie erwartet zwei Argumente; das erste bestimmt die auszuführende Operation, das zweite liefert eine Liste der dabei zu verwendenden Argumentwerte. Folgende Definition kann zugrundegelegt werden:

```
(DEFUN O-EVAL-PRIM-OP (FN VALUES)
   (COND     ((EQ FN 'ATOM) (ATOM (CAR VALUES)))
             ((EQ FN 'CAR) (CAAR VALUES))
             ((EQ FN 'CDR) (CDAR VALUES))
             ((EQ FN 'ZEROP) (ZEROP (CAR VALUES)))
             ((EQ FN 'CONS) (CONS (CAR VALUES) (CADR VALUES)))
             ((EQ FN 'EQ) (EQ (CAR VALUES) (CADR VALUES)))
             ((EQ FN '+) (+ (CAR VALUES) (CADR VALUES)))))
```

In der COND-Form wird die von FN dargestellte primitive Operation ermittelt und dann mit den zugehörigen Argumentwerten ausgeführt. Diese Argumentwerte sind in einer Liste zusammengefaßt, die gegeben ist durch den Wert von VALUES. Liegt nur ein Argument vor, wie bei ATOM , CAR , CDR und ZEROP , so erhält man dessen Wert aus dem Wert von VALUES durch Entfernen des äußeren Klammerpaares, also durch eine Anwendung von CAR. Liegen zwei Argumente vor, wie bei CONS , EQ und + , so ist der Wert des ersten bzw. zweiten Arguments gegeben durch den Wert von (CAR VALUES) bzw. den Wert von (CADR VALUES).

Weshalb wir im Interpreter auch und gerade diejenigen LISP-Funktionen verwenden, die wir zu interpretieren vorgeben, liegt daran, daß wir den LISP-Interpreter eben in LISP beschreiben. Bei einem realen Interpreter steht hier natürlich für jede primitive Operation ein ausführbarer Code, der z.B. in einer Assemblersprache geschrieben sein kann.

Damit haben wir die Funktion O-EVAL-CONS für den Fall der primitiven Operation kennengelernt.

17.2.2.3 Lambda-Ausdrücke und ihre Auswertung

Als nächstes betrachten wir die Auswertung von Lambda-Ausdrücken. Um testen zu können, ob die auszuwertende Funktion in Gestalt eines Lambda-Ausdrucks vorliegt, definieren wir ein Prädikat LAMBDA-P wie folgt:

```
(DEFUN LAMBDA-P (L) (AND (LISTP L) (EQ (CAR L) 'LAMBDA)))
```

Offensichtlich prüft LAMBDA-P , ob das Argument L eine Liste darstellt, die das Symbol LAMBDA als erstes Element enthält.

Bei unserer Konstruktion des Interpreters liegen die auszuwertende Funktion als Wert der Variablen FN vor, die Argumente in Gestalt einer Liste als Wert der Variablen ARGS.

Wir prüfen deshalb mit dem Aufruf

```
(LAMBDA-P FN)
```

ob FN eine Liste darstellt, die das Symbol LAMBDA als erstes Element enthält. Ist dies der Fall, so ist der Rumpf des Lambda-Ausdrucks mit den durch ARGS gegebenen Argumenten in der vorgegebenen Umgebung auszuwerten.
Dazu definieren wir jetzt eine Funktion, die eine Folge von Formen der Reihe nach auswertet und den Wert der letzten Form als Wert übergibt. Wir bezeichnen sie mit O-EVAL-N . Sie soll zwei Parameter erhalten, FORMS und ENV . Über den Parameter FORMS bringen wir eine Liste der auszuwertenden Formen ein, über den Parameter ENV eine A-Liste, mit den Bindungen, unter denen die Formen in FORMS ausgewertet werden sollen.

O-EVAL-N kann nun wie folgt definiert werden:

```
(DEFUN O-EVAL-N (FORMS ENV)
   (COND     ((NULL FORMS) NIL)
             ((NULL (CDR FORMS)) (O-EVAL (CAR FORMS) ENV))
             (T      (O-EVAL (CAR FORMS) ENV)
                     (O-EVAL-N (CDR FORMS) ENV))))
```

Stellt FORMS die leere Liste dar, so wird NIL als Wert übergeben.

Stellt FORMS eine Liste mit einer Form dar, so wird diese ausgewertet und ihr Wert übergeben. Andernfalls wird die erste Form der Liste ausgewertet und dann O-EVAL-N rekursiv auf (CDR FORMS) angewendet.

Die Auswertung des durch den Wert von FN gegebenen Lambda-Ausdrucks, für die durch die Liste ARGS gegebenen Argumente, und die durch eine A-Liste ENV gegebene Umgebung, erreichen wir nun durch den Aufruf

(O-EVAL-N (CDDR FN) (PAIRLIS (CADR FN) (O-EVAL-LIST ARGS ENV) ENV))

O-EVAL-N erwartet, wie wir wissen, als erstes Argument eine Liste, deren Elemente Formen sind, wertet diese unter den vom zweiten Argument - einer A-Liste - beschriebenen Bindungen der Reihe nach aus und übergibt als Wert den Wert der letzten Form. O-EVAL-N erhält hier als erstes Argument (CDDR FN) ; dieses liefert den Rumpf des Lambda-Ausdrucks mit zusätzlichen Klammern, also eine Liste. Das zweite Argument liefert eine A-Liste; diese wird mit PAIRLIS konstruiert; dabei wird die A-Liste für die Umgebung erweitert um die Bindungen der Parameter des Lambda-Ausdrucks, gegeben durch die Elemente von (CADR FN), an die zugehörigen Argumentwerte, gegeben durch die Elemente von (O-EVAL-LIST ARGS ENV).

17.2.2.4 Frei definierte Funktionen und ihre Auswertung mit O-APPLY

Zur Vervollständigung der Definition von O-EVAL-CONS haben wir noch die Auswertung einer durch ein Symbol gegebenen Funktion zu betrachten, die keine spezielle Form (und kein Makro) ist.
Zu ihrer Abarbeitung definieren wir eine Funktion O-APPLY. Wir testen diesen Fall mit dem Prädikat APPLYABLE-P , das wie folgt definiert sei:

```
(DEFUN APPLYABLE-P (FN)
       (AND  (SYMBOLP FN)
             (FBOUNDP FN)
             (FUNCTIONP (SYMBOL-FUNCTION FN))))
```

Wenn der Test (APPLYABLE-P FN) erfüllt ist, wird die Form mit

(O-APPLY (SYMBOL-FUNCTION FN) (O-EVAL-LIST ARGS ENV))

ausgewertet.

Die Funktion O-APPLY ist noch zu definieren. Sie erwartet zwei Argumente: Der Wert des ersten muß ein Lambda-Ausdruck sein; der des zweiten eine Liste, deren Elemente als Argumentwerte für die Auswertung des Lambda-Ausdrucks dienen.
Im obigen Aufruf wird der Lambda-Ausdruck durch Auswerten von (SYMBOL-FUNCTION FN) aus der Funktionszelle von FN geholt. Die Liste der Argumentwerte zur Auswertung des Lambda-Ausdrucks wird mit der uns bereits bekannten Funktion O-EVAL-LIST bestimmt.

Die Funktion O-APPLY kann wie folgt definiert werden:

```
(DEFUN O-APPLY (L-EX VALUES)
        ; Die Parameter L-EX und VALUES stehen für einen Lambda-Ausdruck und
        ; die Liste der aus gewerteten Argumente für diesen
(LET    ((L-PARAMS (CADR L-EX))
         (L-BODY (CDDR L-EX))
        ; hier werden lokale Variable L-PARAMS und L-BODY eingeführt.
        ; L-PARAMS wird gebunden an den Wert von (CADR L-EX), also die
        ; Parameterliste des Lambda-Ausdrucks.
        ; L-BODY wird gebunden an (CDDR L-EX) , also die Liste der Formen,
        ; die den Rumpf des Lambda-Ausdrucks bilden.

 (COND ((EQL    (LENGTH L-PARAMS) (LENGTH VALUES))
                (O-EVAL-N L-BODY (PAIRLIS L-PARAMS VALUES ENV))
        ; wenn die Anzahl der Parameter im Lambda-Ausdruck mit der Anzahl der
        ; Argumentwerte übereinstimmt, wird der Rumpf des Lambda-Ausdrucks in
        ; der durch die PAIRLIS-Form gegebenen Umgebung durch O-EVAL-N
        ; ausgewertet, andernfalls erfolgt eine Fehlermeldung
       (T (ERROR    "WRONG NUMBER OF ARGUMENTS
                     ~%FUNCTION: ~S
                     ~%ARGUMENTS: ~S"
                     L-EX VALUES)))))
```

Zur Auswertung des Rumpfs des Lambda-Ausdrucks haben wir hier wieder die Funktion O-EVAL-N eingesetzt. Das erste Argument L-BODY gibt die Liste der Formen aus dem Funktionsrumpf, das zweite Argument eine Assoziationsliste mit den Bindungen für die in den Formen auftretenden Variablen. Unter diesen Bindungen wertet O-EVAL-N die mit der Liste gegebenen Formen der Reihe nach aus und übergibt als Wert den Wert der letzten Form.

17.2.2.5 Vollständige Auswertung von nichtleeren Listen

Wir haben nun alle Fälle zur Auswertung von nichtleeren Listen behandelt.

Die Funktion O-EVAL-CONS hat jetzt nach Hinzunahme einer Fehlermeldung folgende Gestalt:

```
(DEFUN O-EVAL-CONS (FORM ENV)
  (LET ((FN (CAR FORM))
        (ARGS (CDR FORM)))
       (COND ((SP-FORM-P FN)
              (O-EVAL-SP-FORM (FN ARGS ENV)))
             ((PRIM-OP-P FN)
              (O-EVAL-PRIM-OP FN  (O-EVAL-LIST ARGS ENV)))
             ((APPLYABLE-P FN)
              (O-APPLY  (SYMBOL-FUNCTION FN)
                        (O-EVAL-LIST ARGS ENV)))
             ((LAMBDA-P FN)
              (O-EVAL-N (CDDR FN)
                        (PAIRLIS (CADR FN)
                                 (O-EVAL-LIST ARGS ENV)
                                 ENV)))
             (T    (ERROR "Unknown Function: ~S" FN)))))
```

17.3 Zusammenfassung

Es wurde ein Interpreter für eine Teilmenge von COMMON LISP entwickelt. Er wird beschrieben durch die Funktion O-EVAL (für OUR-EVAL). Sie beschreibt die Auswertung einer Form in einer Umgebung.

Die auszuwertende Form ist	entweder	ein Symbol,
	oder	eine Zahl, ein String, einer der Werte T bzw. NIL
	oder	eine nichtleere Liste.

In den ersten beiden Fällen ist die Auswertung problemlos. Im Falle einer Liste werden die folgenden Unterfälle betrachtet:

Die Liste beschreibt

entweder	eine spezielle Form
oder	eine primitive Operation
oder	eine frei definierte Funktion
oder	einen Lambda-Ausdruck.

Für alle diese Unterfälle werden Auswertungsfunktionen definiert, die insbesondere in der Lage sein müssen, rekursiv in der Listenstruktur abzusteigen und die dabei benötigten Informationen über die jeweils zugrundezulegende Umgebung aufzubauen und korrekt bereitzustellen. Dabei wird auch eine Funktion O-APPLY definiert, eine einfachere Version der uns bekannten Systemfunktion APPLY.

KAPITEL 18:

Arrays

Wenn eine größere Anzahl zusammenhängender Daten, sagen wir Helligkeitswerte für 256 x 512 Bildpunkte, ständig präsent und zugreifbar sein sollen, so können diese nach unseren bisherigen Kenntnissen entweder mit Hilfe von Einzelvariablen oder in Listen organisiert gespeichert werden. Stellen wir uns jetzt aber eine Anwendung vor, bei der ein Bild nach Zerlegen in 256 Zeilen und 512 Spalten durch 256 x 512 Helligkeitswerte dargestellt ist, und bei der auch wechselnde Bildausschnitte (windows) weiterverarbeitet werden sollen. Offensichtlich ist die Darstellung des Bildes durch 256 x 512 unabhängige Einzelvariablen indiskutabel. Aber auch die Speicherung von Daten in einer Liste ist unvernünftig, da man nicht davon ausgehen kann, daß die Einzeldaten im wesentlichen zusammen und stets in derselben Reihenfolge benötigt werden. Andernfalls wäre das System einen Großteil der Zeit nur damit beschäftigt, die Listen "durchzuCDRen". Extrem mühsam wird insbesondere das Verarbeiten von Bildausschnitten (windows).
Um diese Art von "Listenmanipulation" vermeiden zu können, verfügt LISP über die Datenstruktur "Array". Ein Array ist eine Kollektion von Variablen, die alle denselben Namen haben, aber durch ganzzahlige "Indizes" unterschieden werden. Die wichtigste Eigenschaft eines Arrays ist, daß auf jede Komponente gleich schnell zugegriffen werden kann. Jede Komponente eines Arrays kann einen Wert aufnehmen. Um eine Kollektion von 256 x 512 Daten aufnehmen zu können, muß im Speicher ein Array mit 256 x 512 Komponenten eingerichtet werden.

18.1 Einrichtung von Arrays

Arrays werden eingerichtet mit Hilfe der Systemfunktion MAKE-ARRAY.
Ein Aufruf der Gestalt

(MAKE-ARRAY <Dimensionen>)

bewirkt die Einrichtung eines Arrays. Anordnung und Anzahl der Komponenten hängt noch vom Argument Dimensionen ab. Dieses muß als Wert eine Liste von nichtnegativen ganzen Zahlen haben. Diese Zahlen heißen die Dimensionen des Arrays. Die Anzahl der Dimensionen heißt Rang des Arrays.

Beispiel:
Der Aufruf
(MAKE-ARRAY '(5 3 2))
bewirkt die Einrichtung eines Arrays mit drei Dimensionen.Es hat 5 x 3 x 2 Komponenten, die durch drei Indizes, einen für jede Dimension, identifiziert werden. Der Index für die
- erste Dimension 5 kann die Werte 0, 1, 2, 3, 4
- zweite Dimension 3 kann die Werte 0, 1, 2
- dritte Dimension 2 kann die Werte 0, 1

annehmen.

Ein Array mit einer Dimension heißt auch Vektor. Im Spezialfall einer Dimension genügt im Aufruf von MAKE-ARRAY als Wert des Arguments Dimensionen eine nichtnegative ganze Zahl anstelle einer Liste mit dieser Zahl als einzigem Element.

Der folgende Funktionsaufruf bewirkt die Einrichtung eines Arrays mit 2 x 3 Komponenten und die Zuweisung dieser Arraystruktur an das Symbol EXAMPLE :

```
-->(SETQ EXAMPLE (MAKE-ARRAY '(2 3)))
#2A ( . . . )                    ; Wert der Komponenten noch undefiniert
```

Wie bei SETQ üblich, wird der Wert des zweiten Arguments, also der MAKE-ARRAY-Form, als Wert übergeben. Die externe Repräsentation dieses Wertes besteht aus einem Präfix #2A , das anzeigt, daß ein Array mit 2 Dimensionen vorliegt, gefolgt von einer Liste, welche die einzelnen Komponenten des Arrays wiedergibt. Der Wert der Komponenten eines neu erzeugten Arrays ist undefiniert, falls keine explizite Initialisierung vorgenommen wird. Zum Initialisieren können die Schlüsselwortargumente :INITIAL-ELEMENT und :INITIAL-CONTENTS verwendet werden.

Enthält die MAKE-ARRAY-Form das Schlüsselwort :INITIAL-ELEMENT und ein zugehöriges Argument, so wird jede Arraykomponente mit dem Wert dieses Arguments initialisiert.

Mit Hilfe des Schlüsselwortes :INITIAL-CONTENTS und eines geeigneten Arguments können die Arraykomponenten mit verschiedenen Werten initialisiert werden. Der Wert dieses Arguments muß eine geschachtelte Liste sein. Hat das Array den Rang 0, dann spezifiziert der Wert die einzige Komponente. Anderenfalls muß der Wert eine Liste sein, deren Länge mit der ersten Dimension

übereinstimmt. Jedes Listenelement ist wieder eine geschachtelte Liste für ein Array, und entspricht dem Teilarray, dessen Dimensionen mit den verbliebenen Dimensionen übereinstimmen.

Schlüsselwortargumente für :INITIAL-CONTENTS und :INITIAL-ELEMENTS können offensichtlich bei der Anlage eines Feldes nicht gleichzeitig verwendet werden.

Beispiele:

```
-->(MAKE-ARRAY '(2 3) :INITIAL-ELEMENT 0)
#2A ((0 0 0) (0 0 0))

-->(SETQ AR (MAKE-ARRAY '(2 3)
                  :INITIAL-CONTENTS '((A B C) (1 2 3))))
#2A ((A B C) (1 2 3))
```

Die Funktion MAKE-ARRAY hat noch weitere Schlüsselwortargumente:
Wird sie mit dem Schlüsselwort :ELEMENT-TYPE unter Angabe eines Datentyps aufgerufen, so sind als Elemente des Arrays nur Objekte dieses Typs zugelassen. Der Defaultwert T spezifiziert ein Array, dessen Elemente beliebige LISP-Objekte sein können.

Durch Verwendung des Schlüsselwortes :ADJUSTABLE kann man festlegen, daß die Größe des Arrays nach seiner Erzeugung noch dynamisch verändert werden kann. Dazu muß im Aufruf von MAKE-ARRAY zum Schlüsselwort :ADJUSTABLE ein Argument mit einem Wert ungleich NIL angegeben werden.
Bei einem so erzeugten Array kann jede Dimension verändert werden, nicht aber die Anzahl der Dimensionen, d.h. der Rang.

Dimensionen werden verändert mit Hilfe der Funktion ADJUST-ARRAY. Ihr Aufruf entspricht dem von MAKE-ARRAY. Mit Ausnahme von :ADJUSTABLE können auch alle sonst bei MAKE-ARRAY möglichen Schlüsselwortargumente verwendet werden.

Enthält ein ADJUST-ARRAY-Aufruf das Schlüsselwort :INITIAL-ELEMENT, so werden alle neuen Komponenten, die dem alten Array nicht angehörten, mit dem Wert des zugehörigen Arguments initialisiert.
Enthält der ADJUST-ARRAY-Aufruf dagegen das Schlüsselwort :INITIAL-CONTENTS, so legt das zugehörige Argument alle Komponenten des neuen Arrays fest.

Beispiele:

```
-->(SETQ AR (MAKE-ARRAY '(2 3)
                        :INITIAL-CONTENTS '((A B C) (1 2 3))
                        :ADJUSTABLE T))
#2A ((A B C) (1 2 3))

-->(ADJUST-ARRAY AR '(4 2) :INITIAL-ELEMENT 99)
#2A ((A B) (1 2) (99 99) (99 99))

-->(ADJUST-ARRAY AR '(2 4)
                 :INITIAL-CONTENTS '((U V W X) (2 3 5 7)))
#2A ((U V W X) (2 3 5 7))
```

Mit dem Prädikat ADJUSTABLE-ARRAY-P kann festgestellt werden, ob ein Array als "ADJUSTABLE" spezifiziert wurde oder nicht. Dazu dient ein Aufruf der Gestalt

(ADJUSTABLE-ARRAY-P <Array-Name>) .

18.2 Zugriff auf Array-Komponenten, das allgemeine Zuweisungsmakro SETF

Auf die Komponenten eines Arrays kann man mit der System-Funktion AREF zugreifen. Sie wird für ein Array der Dimension r aufgerufen mit dem **Aufrufmuster:**

(AREF <Array-Name> <Index-1> ... <Index-r>)

Das erste Argument bezeichnet ein Array; seine Auswertung eröffnet den Zugriff auf den Speicherbereich des genannten Arrays; die Werte der weiteren Argumente müssen ganze Zahlen aus dem jeweils zugelassenen Indexbereich sein.

Wert:

Die Array-Komponente zu den gegebenen Indexwerten .

Hier ist zu beachten, daß alle Indizes von 0 an laufen.

Beispiel:

```
-->(SETQ AR (MAKE-ARRAY '(2 3) :INITIAL-CONTENTS '((A B C) (1 2 3))))
#2A ((A B C) (1 2 3))

-->(AREF AR 1 2)
3

-->(AREF AR 2 2)
ERROR                                   ; der erste Index kann nur den Wert 0 oder 1
                                        ; annehmen
```

Auch den Komponenten eines bereits erzeugten Arrays können wir Werte zuweisen. Dazu verwenden wir das Zuweisungsmakro SETF.

Beispiel:

```
-->(SETF (AREF AR 1 2) '(A B))
(A B)
```

Durch Abarbeitung dieses Aufrufs erhält im Array AR die Komponente mit den Indexwerten 1 und 2 den Wert (A B).
Der folgende Aufruf bestätigt dies:

```
-->(AREF AR 1 2)
(A B)
```

Dieses Zuweisungsmakro SETF betrachten wir jetzt noch etwas näher. Es wird aufgerufen wie SETQ. Das erste Argument legt fest, wo (als Seiteneffekt !) ein Wert zugewiesen werden soll; dort wird der Wert des zweiten Arguments zugewiesen; er wird auch Wert der SETF-Form.

Verblüffend an obigem Beispiel ist, wie die SETF-Form ihr erstes Argument verwendet:
AREF ist nämlich eine Funktion zum Zugriff auf eine Array-Komponente. Sie liefert für sich allein den Wert dieser Komponente. Die SETF-Form nutzt das Ergebnis der AREF-Auswertung in einer bisher für uns ungewohnten Art und Weise:

SETF ist ein Makro zum Abändern beliebiger Strukturkomponenten; es hat das **Aufrufmuster:**

(SETF <Platzangabe> <Neuer Wert>)

Das erste Argument ist eine Form und gibt an, wo etwas geändert werden soll; hier kann

- eine Variable stehen (in diesem Fall funktioniert SETF wie SETQ) oder
- eine Zugriffsfunktion wie CAR, CDR, GET, AREF oder
- eine Platzangabe ganz anderer Art, darauf gehen wir aber jetzt nicht ein.

Jedenfalls bewirkt SETF, daß der Wert, der normalerweise bei Auswertung der Platzangabe ermittelt wird, an seinem Platz ersetzt wird und zwar durch den Wert des zweiten Arguments. Damit wird zum Beispiel

im Falle einer Variablen A

(SETF A <E>) äquivalent zu (SETQ A <E>)

im Falle einer Zugriffsfunktion wie CAR

(SETF (CAR A) <E>)) äquivalent zu (RPLACA A <E>)

Insbesondere können mit SETF - wie wir gesehen haben - auch ARRAY-Komponenten neu festgelegt werden.

Beispiele:

```
-->(SETQ AR (MAKE-ARRAY '(2 3) :INITIAL-CONTENTS '((A B C) (1 2 3))))
#2A ((A B C) (1 2 3))

-->(AREF AR 0 1)
B

-->(SETF (AREF AR 0 1) 'CAT)
CAT

-->(SETF (AREF AR 1 1) (CAR (A B)))
A
```

```
-->(AREF AR 0 1)
CAT

-->(AREF AR 1 1)
A
```

18.3 Vektoren und Sequenzen

Arrays mit einer Dimension heißen Vektoren. Sie können entweder mit MAKE-ARRAY oder mit einer Funktion VECTOR erzeugt werden. Diese arbeitet ähnlich wie LIST.
Ein Aufruf mit dem Muster

(VECTOR <Arg-1> ... <Arg-n>)

mit $n \geq 1$ Argumenten Arg-1 bis Arg-n

erzeugt einen Vektor mit n Komponenten, deren Werte durch die Werte von Arg-1 bis Arg-n gegeben sind.
Der obige Aufruf ist äquivalent zum Aufruf

(MAKE-ARRAY <Zahl n> :INITIAL-CONTENTS (LIST <Arg-1> ... <Arg-n>))

Beispiel:

```
-->(VECTOR 'A1 'A2 'A3)
#(A1 A2 A3)
```

Die externe Darstellung des Wertes eines Vektors besteht aus dem Präfix # , welches anzeigt, daß hier ein Vektor vorliegt, gefolgt von einer Liste, welche die einzelnen Komponenten des Vektors wiedergibt.

Bei der Erzeugung von Vektoren mit MAKE-ARRAY oder VECTOR kann das Schlüsselwortargument :FILL-POINTER verwendet werden, um einen sogenannten Füllzeiger anzugeben. Das ist eine nichtnegative ganze Zahl, nicht größer als die Gesamtzahl der Vektorkomponenten. Dieser Füllzeiger gibt die Gesamtzahl der "aktiven" oder "eingefüllten" Elemente des Vektors.

Er legt damit die aktive Länge des Vektors fest; aktiv sind alle Komponenten des Vektors, deren Index kleiner ist als der Füllzeiger; die anderen sind inaktiv.

Vektoren haben mit Listen insbesondere eine Eigenschaft gemein: Beide enthalten jeweils eine geordnete Menge von Komponenten.

Es gibt einige Operationen, die für Listen und Vektoren in gleicher Weise sinnvoll sind, weil sie mit der Folge der Komponenten in gleichartiger Weise umgehen. Ein Beispiel ist die Berechnung der Länge mit LENGTH oder die Bestimmung der Umkehrung mit REVERSE bzw. NREVERSE. Deshalb faßt man Listen und Vektoren unter dem Überbegriff "Sequenz" (englisch: sequence) zusammen. Neben den erwähnten Funktionen LENGTH, REVERSE und NREVERSE gibt es noch weitere Funktionen, die für beliebige Sequenzen, d.h. sowohl für Listen als auch für Vektoren, definiert sind.

Wir erwähnen insbesondere die Zugriffsfunktion ELT; sie hat das

Aufrufmuster:
(ELT <-Sequenz-> <Index>)

> Der Wert des ersten Arguments muß eine Sequenz sein, der Wert des zweiten eine nichtnegative ganze Zahl, die kleiner ist als die Länge der Sequenz.

Wert:
Die Komponente der gegebenen Sequenz, die auf der durch den Index gekennzeichneten Position steht. Wieder werden die Positionen von 0 ab gezählt.

Seiteneffekt: Keiner

Mit ELT können demnach sowohl Listenelemente als auch Vektorkomponenten bestimmt werden.

Die Funktion ELT leistet somit bei Vektoren dasselbe wie die Zugriffsfunktion AREF ; sie kann genauso wie AREF in Verbindung mit SETF benutzt werden, um ein Element einer Sequenz destruktiv zu ersetzen.

Beispiele:
```
-->(SETQ L '(A B CD E F)
(A B CD E F)
```

```
-->(SETQ V (VECTOR 1 2 6 24 120 720))
#(1 2 6 24 120 720)

-->(ELT L 2)                          ; Zählen der Positionen von 0 ab
CD

-->(ELT V 4)
120

-->(SETF (ELT V 2) 'CD)
CD

-->(REVERSE V)
#(720 120 24 CD 2 1)
```

18.4 Strings

Eine wichtige Teilklasse der Vektoren bilden die Strings. Sie sind spezielle Vektoren, deren Elemente Zeichen (characters) sind.
Strings unterscheiden sich von Symbolen im wesentlichen dadurch, daß sie in Anführungszeichen eingeschlossen sind. Alle Funktionen, die für Vektoren definiert sind, können auch für Strings verwendet werden. So kann mit AREF oder ELT auf ein Zeichen zugegriffen werden.

Beispiel:

```
-->(AREF "FLOOB-BOOBER" 0)            ; ein String hat sich selbst als Wert
#\F                                   ; Zählung der Positionen beginnt bei 0

-->(ELT "FLOOB-BOOBER" 1)
#\L
```

Das Präfix #\ in der Ausgabe weist darauf hin, daß das nachfolgende Zeichen ein "character" und nicht etwa ein Symbol darstellt.
Daneben gibt es speziell für Strings noch eine Zugriffsfunktion CHAR, die genauso wie AREF und ELT aufgerufen wird, aber nur für Strings verwendet werden kann.

Aufrufmuster:
(CHAR <-String-> <Index>)

Der Wert des ersten Arguments muß ein String sein; da Strings sich selbst als Wert haben, kann das erste Argument auch selbst ein String sein. Der Wert des zweiten Arguments muß eine nichtnegative ganze Zahl sein, die kleiner ist als die Länge des Strings.

Seiteneffekt: Keiner

Beispiel:
```
-->(CHAR "FLOOb-bOOBer" 4)
#\b
```

Auch CHAR kann zusammen mit SETF dazu verwendet werden, ein Zeichen innerhalb eines Strings destruktiv zu verändern.

Beispiel:
```
-->(SETF ST "FLOOb-bOOBer")
"FLOOb-bOOBer"
-->(SETF (CHAR ST 4) B)
#\B
-->ST
"FLOOB-bOOBer"
```

18.5 Vergleich von Strings

Ob zwei Strings übereinstimmen, testet man mit der Funktion STRING= . Sie kann aufgerufen werden mit dem
Aufrufmuster:

(STRING= <-String-1-> <-String-2->)

Jedes der Argumente muß als Wert einen String haben.

Wert:

T falls beide Strings in allen Positionen übereinstimmen, wobei Groß- und Kleinbuchstaben als unterschiedliche Zeichen angesehen werden.

NIL sonst.

Auch Teilstrings können verglichen werden; STRING= verfügt nämlich über vier Schlüsselwortargumente mit den Schlüsselworten

:START1 :START2 :END1 :END2

Die zugehörigen Argumente müssen ganze Zahlen sein. Sie bezeichnen diejenige Position im ersten bzw. zweiten String, mit der der Vergleich beginnt, sowie die Position, vor der der Vergleich endet, also diejenige Position, die als erste nicht mehr verglichen wird. Die Defaultwerte für :START1 und :START2 sind 0, entsprechend einem Vergleich, der mit dem ersten "character" beginnt. Die Defaultwerte für :END1 und :END2 sind die Längen der Strings, entsprechend einem Vergleich bis zum letzten "character".

Beispiel:

```
-->(STRING= "tip" "Tip")
NIL                                          ; verschiedene Strings

-->(STRING= "together" "Frog" :START1 1 :END1 3 :START2 2)
T                                            ; zu vergleichende Teilstrings stimmen
                                             ; überein
```

Die Vergleichsfunktion STRING= unterscheidet zwischen Groß- und Kleinbuchstaben. Eine Modifikation, nämlich STRING-EQUAL, unterscheidet nicht zwischen Groß- und Kleinbuchstaben, arbeitet aber sonst wie STRING=.

Beispiel:

```
-->(STRING-EQUAL "tip" "Tip")
T
```

Wir gehen noch auf die Beziehung zwischen Symbolen und Strings ein. Auf dem Papier läßt sich ein Symbol problemlos durch Anbringen von Anführungszeichen in einen String überführen. Das

LISP-System verwendet dazu die Funktion STRING. Ein Aufruf der Gestalt

(STRING <Symbol>)

liefert als Wert denjenigen String, der aus dem Symbol entsteht, wenn man die Folge seiner "character" als String interpretiert.

Wir sind jetzt in der Lage festzustellen, ob ein Symbol in einer bestimmten Position ein bestimmtes Zeichen hat. Dazu betrachten wir das folgende

Beispiel:
Das Symbol SYM habe als Wert ein anderes Symbol. Um zu prüfen, ob dieses Symbol als zweites Zeichen die Ziffer 4 hat, können wir den folgenden Aufruf verwenden:

```
(STRING= (STRING SYM) "4"      :START1 1  :END1 2
                               :START2 0  :END2 1)
```

Da die für START2 und END2 mitgegebenen Argumente sowieso als Defaultwerte eingesetzt würden, genügt auch folgender **Funktionsaufruf:**

```
(STRING= (STRING SYM) "4"      :START1 1  :END1 2)
```

Es gibt noch weitere Funktionen, die Strings zu manipulieren oder vergleichen, etwa bezüglich ihrer lexikographischen Anordnung. Bei Bedarf sollte man das COMMON-LISP-Manual zu Rate ziehen.

18.6 Informationen über ARRAYS

Gelegentlich ist es wichtig, Informationen über ein Array gewinnen zu können, z.B. über den Rang, die Gesamtzahl der Komponenten oder die Möglichkeit der Veränderung.
Zur Beschaffung solcher Informationen dienen verschiedene Systemfunktionen. Für ein Array als Argument liefert z.B.

ARRAY-RANK	den Rang
ARRAY-TOTAL-SIZE	die Gesamtzahl der Komponenten
ADJUSTABLE-ARRAY-P	die Auskunft, ob das Array als "adjustable" spezifiziert ist.

Beispiel:

```
-->(SETQ AR (MAKE-ARRAY '(9 4 6)))
#3A (...)                          ; da Arraykomponenten undefiniert
-->(ARRAY-RANK AR)
3
-->(ARRAY-TOTAL-SIZE AR)
216
-->(ADJUSTABLE-ARRAY-P AR)
NIL                                ; für :ADJUSTABLE wurde kein
                                   ; Argument mit Wert ungleich NIL
                                   ; angegeben
```

Einige der vielen weiteren Möglichkeiten zur Informationsgewinnung zeigen die
Beispiele:

```
-->(ARRAY-DIMENSION AR 1)          ; bestimmt für das Array die durch das
4                                  ; Argument 1 bezeichnete Dimension,
                                   ; auch hier Zählung von 0 ab

-->(ARRAY-IN-BOUNDS-P AR 8 3 2)    ; entscheidet, ob das Array für einen
T                                  ; Satz von Indizes eine Komponente
                                   ; enthält oder nicht

-->(ARRAY-IN-BOUNDS-P AR 8 6 3)    ; entsprechende Komponente nicht
NIL                                ; vorhanden, da zweite Dimension
                                   ; gleich 4
```

18.7 Zusammenfassung

Arrays sind Datenobjekte mit Komponenten, die über ganzzahlige Indizes angesprochen werden. Ein Array hat null oder mehr Dimensionen. Jede Dimension ist eine nichtnegative ganze Zahl; diese

gibt die Anzahl der möglichen Indexwerte. Alle Indexwerte laufen von 0 an. Ist eine Dimension gleich null, so hat das Array keine Komponente.
Ein Array mit null Dimensionen hat genau eine Komponente. Bei allgemeinen Arrays kann jede Komponente ein beliebiges LISP-Objekt sein. Der Typ der für Komponenten zugelassenen Objekte kann aber auch eingeschränkt werden.
Eindimensionale Arrays heißen auch Vektoren. Vektoren, deren Komponenten Schriftzeichen sind, heißen Strings. Listen und Vektoren faßt man unter dem Sammelbegriff "Sequenzen" zusammen. Viele Funktionen können sowohl für Listen wie für Vektoren verwendet werden.
Vektoren und Arrays gehören genauso wie Strings zu den Atomen, erfüllen also das Prädikat ATOM.

COMMON LISP bietet standardmäßig noch einige spezielle Typen von Arrays und viele Möglichkeiten zu ihrer Manipulation an, die hier nicht behandelt werden konnten.

KAPITEL 19:

Weitere Kontrollstrukturen

Wir wollen in diesem Kapitel nochmals auf Kontrollstrukturen eingehen.
Zuerst betrachten wir verschiedene Möglichkeiten, die Abarbeitung von Programmstücken vorzeitig zu verlassen. Das zu verlassende Programmkonstrukt kann statisch oder dynamisch festgelegt sein. Falls erforderlich kann angeordnet werden, daß sogenannte geschützte (sicherheitsrelevante) Programmteile auf jeden Fall vor dem Verlassen ausgewertet werden.
Anschließend lernen wir einige Wiederholungsanweisungen kennen, die vorteilhaft dann eingesetzt werden, wenn nicht die volle Mächtigkeit von DO benötigt wird.
Zum Abschluß betrachten wir noch die sogenannte Endrekursion, einen Spezialfall der Rekursion, bei dem der sonst übliche Buchhaltungsaufwand weitgehend entfällt, weil letztlich nur eine spezielle Form der Wiederholung vorliegt.

19.1 Ausgänge

19.1.1 Statische Ausgänge aus Blöcken

Wir wissen um die Möglichkeit eine DO-Iteration aufgrund einer RETURN-Form zu verlassen. Dies ist ein Spezialfall eines allgemeinen Konzepts, das uns ermöglicht, die Auswertung eines Programmstücks - eines sogenannten Blocks - unter bestimmten Bedingungen mit einer RETURN-Anweisung zu verlassen.

Grundlage hierfür sind die speziellen Formen BLOCK und RETURN-FROM mit den Aufrufmustern:

(BLOCK <Blockname> <Form-1> ... <Form-n>)
und
(RETURN-FROM <Blockname> <Resultatform>)

Dabei ist Blockname ein Symbol; Form-1 bis Form-n und Resultatform sind Formen.

Zur Auswertung des BLOCK-Konstrukts wird der Blockname nicht ausgewertet; vielmehr werden die Formen Form-1, Form-2 usw. der Reihe nach ausgewertet. Kommt es dabei nicht zur Auswertung einer RETURN-FROM-Form, die den Namen des ausgewerteten Blockes spezifiziert, so wird der Wert der letzten Form, also von Form-n als Wert der BLOCK-Form übergeben. In diesem Fall funktioniert BLOCK genauso wie PROGN. Kommt es jedoch zur Auswertung einer RETURN-FROM-Form, die denselben Blocknamen spezifiziert, so wird nur noch deren Resultatform ausgewertet und als Wert der BLOCK-Form übergeben. Damit ist die Auswertung der BLOCK-Form abgeschlossen.

Übrigens wird bei jeder Definition einer Funktion mit DEFUN der Funktionsrumpf implizit als Block spezifiziert und zwar mit dem für die Funktion definierten Namen. Die Auswertung einer mit DEFUN definierten Funktion kann also mit RETURN-FROM vorzeitig beendet werden.

Beispiel:
Die Untersuchung von Speiseölen auf unerlaubte Zusätze soll automatisiert werden. Wir definieren dazu eine Funktion BELASTUNG mit den Parametern OEL (zur Identifikation des Öls), STOFFE (für eine Liste der relevanten Schadstoffe) und GRENZEN (für eine Liste der maximal zulässigen Konzentrationen).
Sie ermittelt den jeweiligen Schadstoffgehalt durch Aufruf einer Funktion ANTEIL (sie wird hier nicht definiert) mit den Parametern OEL und SCHADSTOFF und bricht die Untersuchung ab, sobald für einen Schadstoff eine unzulässige Konzentration festgestellt ist.

```
(DEFUN BELASTUNG (OEL STOFFE GRENZEN)
   (FORMAT NIL "~%Ein Liter ~S enthält" OEL)
   (DO ((L STOFFE (CDR L))
        (G GRENZEN (CDR G)))
       ((EQ L NIL) "Nicht unzulässig belastet")
       (SETQ K (ANTEIL OEL (CAR L)))
       (FORMAT T "~%~S ~15T vertreten mit ~A ~15T
                  wobei Grenzwert ~A" (CAR L) K (CAR G))
       (IF (> K (CAR G))
                  (RETURN-FROM BELASTUNG "Belastung zu hoch"))))
```

Auch der Rumpf einer DO-Form ist implizit als Block definiert, allerdings ohne Namen; wir sagen auch, er hat den Namen NIL. Wie aber kann so ein "namenloser" Block verlassen werden ? Die Antwort ist einfach: durch Auswerten von

(RETURN-FROM NIL <Resultatform>)

Wir haben dazu allerdings bisher den Aufruf

(RETURN <Resultatform>)

verwendet.

Tatsächlich ist RETURN ein Makro, das vom LISP-System zur Verfügung gestellt wird, wobei folgende Definition zugrundeliegt:

```
(DEFMACRO RETURN (RESULT)
    (RETURN-FROM NIL ,RESULT))
```

19.1.2 Dynamische Ausgänge

Bei Verlassen eines Blocks aufgrund von RETURN-FROM oder RETURN liegt der zu verlassende Block stets von vorneherein im Programm fest. Daneben bietet COMMON LISP auch die Möglichkeit, daß die zu verlassende Programmeinheit erst während der Laufzeit des Programms dynamisch ermittelt wird.

Dazu dienen die speziellen Formen CATCH und THROW. Sie haben die
Aufrufmuster:

(CATCH <Ziel> <Form-1> ... <Form-n>)
und
(THROW <Ziel> <Resultatform>)

> Die Argumente Ziel, Form-1 bis Form-n und Resultatform sind Formen; insbesondere Ziel hat als Wert ein beliebiges LISP-Objekt.

Zur Auswertung einer CATCH-Form wird zuerst das Argument Ziel ausgewertet und sein Wert als Identifikation der CATCH-Form eingeführt. Diese Identifikation der CATCH-Form wird gespeichert. Sodann werden die Argumente Form-1, Form-2 usw. der Reihe nach ausgewertet und der Wert von Form-n als Wert der CATCH-Form übergeben, wenn nicht während der Auswertung einer dieser Formen eine THROW-Form ausgewertet wird, deren Ziel mit dem Ziel der

CATCH-Form bezüglich EQ übereinstimmt. In diesem Fall wird die Auswertung von Argumenten der CATCH-Form gestoppt; es wird nur noch die Resultatform der THROW-Form ausgewertet und ihr Wert als Wert der CATCH-Form übergeben. Zugleich wird die jetzt nicht mehr weiter benötigte Identifikation der CATCH-Form aufgegeben.
Das Argument Ziel stellt die Beziehung zwischen THROWs ("Werfern") und CATCHes ("Fängern") her. Es ist ein Fehler, wenn es zur Auswertung einer THROW-Form kommt, aber keine "passende" CATCH-Form zum "Fangen" vorhanden ist.

Beispiel:
Das folgende Programmstück illustriert CATCH und THROW:

```
(SETQ AUSGANG 'EXIT)
(DEFUN KO (LISTE)
   (CATCH AUSGANG
          (APPLY #'+ (CATCH 'ANOTHER-TAG
                       (MAPCAR #'KO-HILF LISTE)))))

(DEFUN KO-HILF (X)
   (IF  (< X 0)
        (THROW 'EXIT "Meßgerät defekt")
        (* 4 3.14 (+ X 1.5))))
```

Die Funktion KO berechnet aus einer Liste von Meßwerten einen Kontrollwert. Der Rumpf von KO enthält zwei ineinandergeschachtelte "CATCHes"; die innere CATCH-Form enthält eine MAPCAR-Wiederholung für eine Hilfsfunktion KO-HILF, in deren Rumpf ein THROW auftritt, das bei Meßwerten kleiner 0 einen entsprechenden Resultatwert "Meßgerät defekt" zum zugehörigen CATCH befördert. Man beachte, daß dieser THROW durch das innere CATCH nicht gefangen wird.

19.1.3 UNWIND-PROTECT zur Sicherung von Seiteneffekten bei THROWS

Gelegentlich muß sichergestellt sein, daß nach Auswerten einer Form noch gewisse Seiteneffekte ausgeführt werden. Ein typisches Beispiel ist

```
(PROGN (START-MOTOR)
       (DRILL-HOLE)
       (STOP-MOTOR))
```

Angenommen im Zuge der Abarbeitung von DRILL-HOLE wird ein THROW abgearbeitet, das zu einem CATCH außerhalb der PROGN-Form führt (weil vielleicht der Bohrer gebrochen ist), so wird (STOP-MOTOR) nicht mehr ausgewertet.

Dies kann vermieden werden durch Verwenden einer UNWIND-PROTECT-Form. Sie hat das **Aufrufmuster:**

(UNWIND-PROTECT <geschützte-Form> <Form-1> ... <Form-n>)

mit den Argumenten geschützte-Form und Form-1 bis Form-n .

Auswertung von UNWIND-PROTECT:
Die Argumentformen werden der Reihe nach ausgewertet; zuerst geschützte-Form , ihr Wert gibt auch den Wert der UNWIND-PROTECT-Form , dann die weiteren Formen wegen ihrer Seiteneffekte. Dabei ist die Auswertung der weiteren Formen grundsätzlich garantiert, unabhängig davon, ob die geschützte Form normal abgearbeitet wird oder aufgrund eines THROW oder RETURN-FROM verlassen werden muß. Die Übergabe an den CATCHer oder den zugehörigen Block wird aufgeschoben, bis die weiteren Formen der UNWIND-PROTECT-Form ausgewertet sind.

Beispiel:

Unser obiges Beispiel zur Steuerung einer Bohrmaschine kann mit UNWIND-PROTECT wie folgt korrekt formuliert werden.

```
(UNWIND-PROTECT
   (PROGN     (START-MOTOR)
              (DRILL-HOLE))
   (STOP-MOTOR))
```

Kommt es hier beim Abarbeiten der PROGN-Form, also während der Abarbeitung von START-MOTOR oder DRILL-HOLE zu einem THROW oder RETURN-FROM, so wird vor dem Verlassen der UNWIND-PROTECT-Form noch (STOP-MOTOR) abgearbeitet.

Wir haben hier allerdings angenommen, daß die Abarbeitung von (STOP-MOTOR) auch dann zulässig ist, wenn der Motor noch nicht gestartet wurde. Es könnte ja sein, daß eine THROW- oder RETURN-Anweisung abzuarbeiten ist, bevor (START-MOTOR) abgeschlossen ist.

19.2 Einfache Wiederholungen

Die DO-Form ist - wie wir wissen - ein sehr mächtiges und allgemeines Konzept für die Iteration. Für spezielle Formen der Wiederholung stellt COMMON LISP noch einfacherere Konzepte zur Verfügung. Wir betrachten zuerst zwei Varianten von DO, nämlich die Makros DOLIST und DOTIMES: Beide verwenden nur eine Laufvariable, diese durchläuft schrittweise die Elemente einer vorgegebenen Liste bzw. bei Vorgabe einer ganzen Zahl n die Zahlen von 0 bis n-1 und wertet bei jedem Schritt ihren Rumpf aus.

DOLIST hat das **Aufrufmuster:**

(DOLIST (<Variable> <Listenform> <Resultatform>) <Rumpf>)

Das erste Argument ist eine Liste. Ihr erstes Element ist ein Symbol, das als Laufvariable dient; das zweite Element ist eine Form, deren Wert eine Liste ist; über diese Liste läuft dann die Laufvariable; das dritte Element ist optional; es ist eine auswertbare Form. Der Rumpf besteht aus einer Folge von null oder mehr auswertbaren Formen.

Auswertung:
Zuerst wird die Listenform ausgewertet; sie liefert eine Liste. Dann wird die Laufvariable der Reihe nach an die einzelnen Elemente dieser Liste gebunden und dazu jeweils der Rumpf ausgewertet. Abschließend wird die Resultatform - soweit vorhanden - ausgewertet.

Wert
der DO-Form wird der Wert der Resultatform, falls eine solche vorhanden ist, bzw. NIL, falls keine Resultatform vorhanden ist.

Seiteneffekt:
Neben den mit der Auswertung des Rumpfes und der Resultatform eventuell verbundenen Seiteneffekten keine weiteren Seiteneffekte.

Beispiel:

Unter der Voraussetzung, daß die globale Variable *STANDARD-OUTPUT* den Ausgabestrom zum Benutzerterminal repräsentiert (dies ist der übliche Defaultwert) betrachten wir die Auswertung von

```
-->(DOLIST (X '(A B C D))
           (PRIN1 X)
           (PRINC " "))
A B C D
NIL
```

Zur Auswertung wird X zuerst an A, dann an B, dann an C und schließlich an D gebunden. Für jede dieser Bindungen wird der Rumpf ausgewertet. Dabei wird als Seiteneffekt der jeweilige Wert von X mit einem anschließenden Leerzeichen ausgegeben. Da keine Resultatform vorhanden ist, wird zum Schluß noch NIL als Wert der DOLIST-Form ausgegeben.

Häufig treten auch Wiederholungen auf, bei denen die Laufvariable über eine Folge von natürlichen Zahlen läuft. Sie können im Prinzip immer mit einem DOLIST-Aufruf der folgenden Gestalt erfaßt werden:

(DOLIST (<Variable> '(0 1 2 ... n-1) <Resultatform>) <Rumpf>)

Da dieser Spezialfall sehr häufig benötigt wird, steht hierfür in COMMON LISP ein eigenes Makro, nämlich DOTIMES zur Verfügung.Ein Aufruf der Gestalt

(DOTIMES (<Variable> <ganze Zahl n> <Resultatform>) <Rumpf>)

wird abgearbeitet wie der angegebene DOLIST-Aufruf mit der Laufliste (0 1 2 ... n-1).

Beispiel:
Eine Funktion zur Berechnung der Fakultät kann wie folgt definiert werden:

```
(DEFUN FACULTAET (N)
   (LET ((F 1))
        (DOTIMES (VAR N F)
          (SETQ F (* F (+ VAR 1))))))
```

Das einfachste Sprachkonstrukt zur Durchführung von Wiederholungen überhaupt ist das Makro LOOP mit dem **Aufrufmuster:**

(LOOP <Form-1> <Form-2> ... <Form-n>), wobei $n \geq 0$.

Auswertung:
Das Makro LOOP verwendet keine Iterationsvariable. Es hat n ≥ 0 Formen als Argumente. Diese dürfen keine Atome sein. Zur Auswertung von LOOP werden diese Formen der Reihe nach ausgewertet; zuerst Form-1, dann Form-2 usw. Wenn die letzte Form Form-n ausgewertet ist, dann beginnt die Auswertung von Form-1, Form-2, usw. von neuem.
Die Auswertung der LOOP-Form bricht also nicht von selbst ab, sie muß vielmehr explizit abgebrochen werden, zum Beispiel mit RETURN oder THROW. LOOP richtet genauso wie die anderen bisher erwähnten Formen der Wiederholung implizit einen Block mit dem Namen NIL ein. Somit kann etwa ein (RETURN <Resultatform>) benutzt werden, um ein LOOP mit einem festgelegten Resultatwert zu verlassen.

Seiteneffekt:
Nur die mit der Auswertung der Argumentformen verbundenen Seiteneffekte.

19.3 Iteration und Endrekursion

Rekursive Funktionen überraschen immer wieder durch ihre elegante, knappe und für den Geübten schnell überschaubare Beschreibung der Lösung. Der Preis hierfür kann jedoch mitunter recht hoch werden.
Die Auswertung einer rekursiven Funktion fordert im allgemeinen beträchtlichen Verwaltungsaufwand, weil über die rekursiven Aufrufe und die Weiterverarbeitung ihrer Ergebnisse Buch geführt werden muß. Die Notwendigkeit zu dieser Buchführung entfällt allerdings, wenn das Ergebnis des Funktionsaufrufs unmittelbar als Ergebnis des rekursiven Funktionsaufrufs anfällt.
Diese Eigenschaft finden wir zum Beispiel bei der Funktion OUR-MEMBER mit der Definition:

```
(DEFUN OUR-MEMBER (EXPR LST)
   (COND  ((NULL L) NIL)
          ((EQL EXPR (CAR LST)) LST)
          (T (OUR-MEMBER EXPR (CDR LST)))))
```

Bei OUR-MEMBER besteht die Wirkung des rekursiven Funktionsaufrufes ausschließlich darin, den gegebenen Funktionsaufruf durch einen Aufruf derselben Funktion für einfachere Argumente zu ersetzen. Offensichtlich wird das Ergebnis eines rekursiven Aufrufs nicht weiter verarbeitet, sondern unmittelbar als Ergebnis des ursprünglichen Aufrufs angesehen. Damit entfällt jede Buchhaltung über die Weiterverarbeitung der Ergebnisse der rekursiven Aufrufe.

Rekursive Funktionen mit dieser Eigenschaft nennen wir endrekursiv.
Eine Funktion heißt **endrekursiv (tail recursive)**, wenn der Wert eines Funktionsaufrufs entweder direkt oder als Wert eines rekursiven Aufrufs berechnet wird.
Aufgrund der stark vereinfachten Buchhaltung können endrekursive Funktionen sehr effektiv abgearbeitet werden. Voraussetzung dafür ist allerdings, daß die Eigenschaft der Endrekursivität vom System erkannt wird.

Darüber hinaus können endrekursive Funktionen sogar automatisch in äquivalente nichtrekursive Funktionen transformiert werden. Viele LISP-Systeme führen diese Transformation intern durch und ermöglichen damit eine sehr effiziente Auswertung endrekursiver Funktionen.

Wir betrachten zwei Beispiele für Operationen auf Listen, die relativ einfach mit endrekursiven Funktionen realisiert werden können.

Beispiel 1:
Die folgende Funktion schneidet die ersten N Elemente einer Liste L ab und übergibt den Rest als Wert

```
(DEFUN FIRST-N-OFF (L N)
   (COND  ((ZEROP N) L)
          (T (FIRST-N-OFF (CDR L) (- N 1)))))
```

Im Falle N = 0 wird von der Liste nichts abgeschnitten. Anderenfalls, also für N > 0, wird die Funktion rekursiv aufgerufen, um von der um 1 Element verkürzten Liste noch weitere N-1 Elemente abzuschneiden.

Die Funktion FIRST-N-OFF ist offensichtlich endrekursiv; der Wert eines Funktionsaufrufs wird im Falle N = 0 direkt und im Falle N > 0 als Wert eines rekursiven Aufrufs berechnet.

Wir hätten die gestellte Aufgabe auch mit einer iterativ arbeitenden Funktion lösen können. Dazu betrachten wir folgende Definition.

```
(DEFUN FIRST-N-OFF-IT (L N)
      (DOTIMES (VAR N)
         (SETQ L (CDR L))))
```

Die Funktion FIRST-N-OFF-IT entfernt iterativ die ersten N Elemente der Liste L und übergibt den Rest als Wert.

Beispiel 2:
Wir versuchen jetzt eine Funktion zu definieren, welche aus den ersten N Elementen einer Liste L eine neue Liste bildet und diese als Wert übergibt. Ein erster Versuch liefert folgende Funktionsdefinition:

```
(DEFUN FIRST-N (L N)
   (COND ((ZEROP N) NIL)
         (T (CONS (CAR L) (FIRST-N (CDR L) (- N 1))))))
```

Im Falle N = 0 ist die Resultatliste leer.
Im Falle N > 0 werden das erste Listenelement und vom Rest der Liste die ersten N-1 Elemente zur Resultatliste zusammengefaßt.

Die Funktion FIRST-N ist nicht endrekursiv; der von einem rekursiven Aufruf übergebene Wert wird als Argument für CONS verwendet. Zudem muß für die Auswertung von CONS der Wert von (CAR L) gespeichert werden, bis der rekursive Aufruf ausgewertet ist.

Im folgenden beschreiben wir eine Lösung mit Hilfe einer endrekursiven Hilfsfunktion:

```
(DEFUN FIRST-N-TAIL-RECURSIVE (L N)
   (AUX L NIL N))
(DEFUN AUX (L RESULT N)
   (COND ((ZEROP N) (REVERSE RESULT))
         (T (AUX (CDR L) (CONS (CAR L) RESULT) (- N 1)))))
```

Im Rumpf von FIRST-N-TAIL-RECURSIVE wird die endrekursive Hilfsfunktion AUX aufgerufen; sie hat 3 Parameter.
Zweck der Hilfsfunktion AUX ist die Einführung einer Hilfsvariablen - hier RESULT genannt - mit der die gesuchte Liste (zunächst invertiert) aufgebaut wird.

Im Falle N = 0 gibt die Umkehrung von RESULT direkt das Resultat an.
Im Falle N > 0 wird AUX rekursiv mit veränderten Parametern aufgerufen. Dazu wurde vom ersten Parameter, der Liste L, das erste Listenelement entfernt und an den zweiten Parameter, nämlich RESULT, vorne "angeCONST*"; gleichzeitig wurde der Parameter N um 1 erniedrigt.

Das Resultat des ursprünglichen Aufrufs wird für N = 0 direkt berechnet und ist im Falle N > 0 identisch mit dem Resultat des rekursiven Aufrufs. Somit ist AUX endrekursiv.
Auch hier geben wir eine äquivalente iterativ arbeitende Funktion an:

```
(DEFUN FIRST-N-LIST (L N)
  (LET ((RESULT NIL))
       (DOTIMES (VAR N)
         (SETQ  RESULT (CONS (CAR L) RESULT)
                       L (CDR L)))    ; mehrfache Wertzuweisung
       (REVERSE RESULT)))
```

Die Funktion FIRST-N-LIST arbeitet offensichtlich genauso wie die Funktion FIRST-N-TAIL-RECURSIVE, nur mit iterativer Umschreibung der Endrekursion.

Wir haben jetzt zwei Beispiele für endrekursive Funktionen gesehen, die relativ einfach in äquivalente iterativ arbeitende Funktionen überführt werden können. Dies war kein Zufall. Endrekursive Funktionen können immer und sogar automatisch in äquivalente nichtrekursive Funktionen transformiert werden. Viele LISP-Systeme führen diese Transformation intern durch und ermöglichen damit eine sehr effiziente Auswertung endrekursiver Funktionen.

19.4 Zusammenfassung

Blöcke sind Programmstücke, deren Abarbeitung mit RETURN-FROM unter Übergabe eines festgelegten Resultatwertes abgebrochen werden kann.
Sie können explizit als benannte Blöcke definiert sein. Bei vielen Sprachkonstrukten, die einen Rumpf haben, ist dieser implizit als benannter Block (wie bei Funktionen) oder als unbenannter Block (wie bei allen bisher eingeführten Wiederholungsanweisungen) definiert. Zum Verlassen unbenannter Blöcke genügt auch die RETURN-Anweisung. Mit CATCH und THROW kann das zu verlassende Programmstück auch dynamisch festgelegt werden. Teile eines zu verlassenden Programmstücks können mit UNWIND-PROTECT so "geschützt werden", daß sie auf jeden Fall vor dem Verlassen des Programmstücks noch ausgewertet werden.

Wenn die volle Mächtigkeit der DO-Iteration nicht benötigt wird, ist es sinnvoll, einfache Iterationsanweisungen wie DOLIST und DOTIMES oder auch LOOP zu verwenden.

Die Endrekursion beschreibt einen Spezialfall der Rekursion, der nur eine einfache Form der Wiederholung darstellt und sehr effektiv abgearbeitet werden kann.

KAPITEL 20:

Mustervergleich (Pattern-Matching)

Wir betrachten in diesem Kapitel die Anwendung von LISP zur Lösung einer Grundaufgabe der Künstlichen Intelligenz.
Diese Grundaufgabe besteht darin, für eine Datenstruktur und ein Muster (engl. pattern) festzustellen, ob die Datenstruktur dem Muster entsprechend (engl. matching) aufgebaut ist.

Man spricht vom "Pattern matching" oder auch vom "Mustervergleich". Diesem "Pattern matching" kommt in vielen Bereichen der Künstlichen Intelligenz eine Schlüsselrolle zu, unter anderem

- bei der Abarbeitung von PROLOG-Programmen (PROLOG ist nach LISP sicher die zweitwichtigste Programmiersprache der Künstlichen Intelligenz,
- beim automatischen Beweisen und
- bei regelbasierten Expertensystemen.

Das Vorgehen kann dabei jeweils wie folgt beschrieben werden: Aus gegebenen Aussagen ("Fakten") - sie sind dargestellt durch Zeichenfolgen - sollen Schlüsse gezogen , also neue Aussagen ("Konsequenzen") gewonnen werden.
Die möglichen Schlußweisen sind festgelegt durch "Produktionen" der Gestalt:

Muster A ----------> Muster B

Eine solche Produktion ist anwendbar auf ein Faktum F, wenn dieses nach dem Muster A aufgebaut ist. Dies wird durch Mustervergleich festgestellt. Die Anwendung liefert ein neues Faktum, das durch die rechte Seite der Produktion und eventuell beim Mustervergleich gewonnene Information bestimmt ist.

Wir gehen hier davon aus, daß alle Fakten durch LISP-Objekte dargestellt sind. Auch die Muster haben die Gestalt von LISP-Objekten, aber mit Platzhaltesymbolen, die Teilstücken des Faktums entsprechen können.
Dabei ist es für viele Anwendungen wichtig, daß bei erfolgreichem Vergleich auch festgehalten wird, welche Teile des Faktums den verschiedenen Platzhaltesymbolen im Muster zugeordnet wurden.
Beim Mustervergleich können für verschiedene Platzhaltesymbole unterschiedliche Anforderungen an die Übereinstimmung beim Vergleich gestellt werden.

Wir beginnen mit einfach strukturierten Mustern und sehr engen Vergleichskriterien und kommen dann zu allgemeineren und flexibleren Vergleichskriterien, bei denen als Resultat auch Information über die gefundene Korrespondenz zwischen Faktum und Muster übergeben wird.

20.1 Einfacher Mustervergleich

Wir betrachten zunächst den Vergleich eines Musters mit einem Datenobjekt für den Fall, daß beide durch Listen von Symbolen dargestellt sind. Die Elemente des Musters sind entweder fest vorgeschrieben - wir sprechen von Musterkonstanten - oder können ein beliebiges Symbol bzw. eine Folge von beliebigen Symbolen vertreten. Allgemein dient

? als Platzhalter für ein beliebiges Symbol

+ als Platzhalter für eine Folge von ein oder mehr Symbolen.

Wir nehmen an, eine Datenbank enthalte 5 Datenobjekte für gewisse Aussagen.

(George is-father-of Bill)
(George is-grandfather-of Jane)
(Karen is-mother-of Tom)
(Susan is-mother-of Tim)
(George is-father-of Susan)

Um festzustellen,
ob George der Vater von Alice ist, genügt es zu klären, ob das Muster

(George is-father-of Alice)

mit einem Datenobjekt aus der Datenbank exakt übereinstimmt.

Diese Übereinstimmung zwischen einem Muster und einem Objekt wird getestet mit einer Funktion MATCH gemäß folgender Definition:

```
(DEFUN MATCH (PA OB)
  (COND   ((AND (NULL PA) (NULL OB)) T)
          ((OR (NULL PA) (NULL   OB)) NIL)
          ((EQUAL (CAR PA) (CAR OB))
            (MATCH (CDR PA) (CDR OB)))))
```

Die Parameter PA und OB übernehmen als Argumente das Muster (Pattern) und das damit zu prüfende Objekt. Falls PA und OB zugleich die leere Liste darstellen, liegt Übereinstimmung vor; dementsprechend wird der Wert T übergeben. Stellt nur eines der Argumente die leere Liste dar, so

liegt keine Übereinstimmung vor. Stimmen die CAR-Teile von PA und OB überein, so liegt Übereinstimmung nur vor, wenn auch die CDR-Teile übereinstimmen. Dies wird mit dem rekursiven Aufruf überprüft.

Um festzustellen,
ob George in der Datenbank als Vater erscheint, genügt es zu klären, ob das Muster

(George is-father-of ?)

in dem ? Platzhalter für ein beliebiges Symbol ist, mit einem Objekt in der Datenbank übereinstimmt.

Um diesen allgemeineren Mustervergleich durchführen zu können, erweitern wir die Funktion MATCH:
Falls das Muster irgendwann als erstes Listenelement ein ? aufweist, paßt das Objekt zum Muster, wenn für die beiden CDR-Teile Übereinstimmung hergestellt werden kann. Die entsprechende Definition von MATCH lautet dann

```
(DEFUN MATCH (PA OB)
  (COND   ((AND (NULL PA) (NULL OB)) T)
          ((OR (NULL PA) (NULL OB)) NIL)
          ((OR (EQUAL (CAR PA) '?)
               (EQUAL (CAR PA) (CAR OB)))
           (MATCH (CDR PA) (CDR OB)))))
```

Um festzustellen,
ob George als Eltern- oder Großelternteil eines Kindes in der Datenbank geführt wird, kann jetzt ein Vergleich mit dem Muster

(George ? ?)

durchgeführt werden.

Um auch einen Vergleich mit dem Muster

(George +)

durchführen zu können, schließlich haben wir + als Platzhalter für eine Folge von ein oder mehr Symbolen eingeführt, erweitern wir die Definition der Funktion MATCH nochmals zu

```
(DEFUN MATCH (PA OB)
  (COND   ((AND (NULL PA) (NULL OB)) T)
          ((OR (NULL PA) (NULL OB)) NIL)
```

```
((OR (EQUAL  (CAR PA) '?)
     (EQUAL (CAR PA) (CAR OB)))
  (MATCH (CDR PA) (CDR OB)))
 ((EQUAL (CAR PA) '+)
  (OR (MATCH (CDR PA) (CDR OB))
      (MATCH PA (CDR OB))))))
```

20.2 Patternvariable

Beim Mustervergleich, wie oben beschrieben, wird festgestellt, ob ein Objekt einem Muster entspricht oder nicht.
Gerade bei einem erfolgreichen Mustervergleich ist es häufig wichtig zu wissen, welche Teile des Objekts auf die Platzhalter abgebildet wurden. Um diese Information speichern und benutzen zu können, führen wir jetzt in Ergänzung zu den Platzhaltern ? und + sogenannte Patternvariable ein. Jeder Patternvariablen wird dann bei erfolgreichem Vergleich derjenige Teil des Objektes zugeordnet, der ihr beim Vergleich entsprach. Diese Zuordnungen werden gesammelt in einer A-Liste, die von MATCH als Wert übergeben wird.

Wir vereinbaren Patternvariable durch Symbole darzustellen, die mit dem Gleichheitszeichen = oder dem Pluszeichen + beginnen. Beispiele sind

=CHILD, =X bzw. +REST, +ETWAS

Es sei darauf hingewiesen, daß diese Patternvariablen keine Variablen im Sinne von LISP sind.

Natürlich müssen wir in der Lage sein, für ein Symbol festzustellen, ob es eine solche Patternvariable darstellt oder nicht.
Dazu definieren wir die Prädikate **S-PATTVAR-P UND M-PATTVAR-P** wie folgt:

```
(DEFUN S-PATTVAR-P (SYMBOL)
  (STRING= (STRING SYMBOL) "="  :END1  1))

(DEFUN M-PATTVAR-P (SYMBOL)
  (STRING= (STRING SYMBOL) "+"  :END1  1))
```

Diese Prädikate überprüfen, ob der aus dem Symbol gebildete String (STRING SYMBOL) mit dem String "=" bzw. "+" in der Position 0 übereinstimmt.

Wir vereinbaren, daß die in Mustern auftretenden Patternvariablen für den Vergleich selbst wie ? bzw. + wirken sollen. Zusätzlich aber soll bei erfolgreichem Vergleich jeder Patternvariablen derjenige Teil des Objektes zugeordnet werden, der ihr bei erfolgreichem Vergleich entsprach.

Diese Zuordnungen sollen in einer A-Liste übergeben werden.
Dabei soll der Aufruf

```
->(MATCH  '(=FATHER IS-FATHER-OF =CHILD)
          '(BILL IS-FATHER-OF FRED))
```

als Wert die A-Liste

```
((=FATHER BILL) (=CHILD FRED))
```

liefern.

Analog soll der Aufruf

```
->(MATCH  '(+INFO FRED)
          '(BILL IS-FATHER-OF FRED))
```

als Wert die A-Liste

```
((+INFO BILL IS-FATHER-OF))
```

liefern.

20.3 Eine Funktion MATCH

Die Funktion MATCH soll jetzt so erweitert werden, daß Patternvariable der betrachteten Gestalt verwendet werden können.

```
(DEFUN MATCH (PA OB &OPTIONAL A-LIST)
  (COND ((AND (NULL PA) (NULL OB))
         (COND ((NULL  A-LIST) T) (T A-LIST)))
        ((OR (NULL PA) (NULL OB)) NIL)
        ((OR (EQUAL (CAR PA) '?)  (EQUAL (CAR PA) (CAR OB)))
         (MATCH (CDR PA) (CDR OB) A-LIST))
```

```
((EQUAL (CAR PA) '+)
 (OR  (MATCH (CDR PA) (CDR OB) A-LIST)
      (MATCH PA (CDR OB) A-LIST)))
((S-PATTVAR-P (CAR PA))
 (MATCH  (CDR PA) (CDR OB)
         (APPEND A-LIST (LIST (LIST (CAR PA) (CAR OB))))))
((M-PATTVAR-P (CAR PA))
 (LET  ((N-A-LIST (EXTEND A-LIST  (CAR PA)  (CAR OB))))
       (OR (MATCH (CDR PA) (CDR OB) N-A-LIST)
           (MATCH PA (CDR OB) N-A-LIST))))))
```

Wir haben hier die Funktion MATCH noch mit einem optionalen Parameter A-LIST mit dem Defaultwert NIL versehen. Er übernimmt in rekursiven Aufrufen von MATCH die mit den bereits ermittelten Bindungen der Patternvariablen gebildete A-Liste. Bei erfolgreichem Mustervergleich übergibt MATCH den Wert T, falls die A-Liste leer ist, ansonsten aber die A-Liste.
Wenn eine Patternvariable, deren P-Name mit dem Zeichen = beginnt, durch das Prädikat S-PATTVAR-P als erstes Element eines Musters erkannt wird, so wird unter zusätzlicher "Assoziation" der Patternvariablen an das erste Element des Objekts ein Abgleich der Reste von Pattern und Objekt angestrebt. Die Assoziation der Patternvariablen wird dabei am Ende der bestehenden A-Liste eingetragen. Ähnlich wird vorgegangen, wenn eine Patternvariable, deren P-Name mit dem Zeichen + beginnt, durch das Prädikat M-PATTVAR-P erkannt wird: Doch muß hier beachtet werden, daß die Patternvariable entweder an ein Objektelement oder an eine Folge von mehreren Objektelementen gebunden werden kann. Diese Assoziation ließ sich im Falle der einfachen Patternvariablen leicht in die jeweilige A-Liste aufnehmen. Im Falle der mehrfachen Patternvariablen verwenden wir für die Aufnahme einer Assoziation in die A-Liste eine Funktion EXTEND, die auch berücksichtigt, daß die Assoziation für eine mehrfache Patternvariable in Schritten aufgebaut werden muß.

Die Funktion EXTEND wird wie folgt definiert:

```
(DEFUN EXTEND (A-LIST PATTVAR ITEM)
  (COND    ((NULL A-LIST) `((,PATTVAR ,ITEM)))
           ((EQUAL PATTVAR (CAAR A-LIST))
            (CONS `(,PATTVAR ,@(APPEND (CDAR A-LIST) (LIST ITEM)))
                  (CDR A-LIST)))
           (T (CONS (CAR A-LIST)
                    (EXTEND (CDR A-LIST) PATTVAR ITEM)))))
```

Ergänzungen der A-Liste und Neueinträge werden nur am Ende vorgenommen. Dazu wird die A-Liste rekursiv durchgearbeitet, bis entweder nur noch die leere Liste übrig ist, oder die Patternvariable als CAR-Komponente eines Eintrags gefunden wird. Im ersten Fall wird eine neue Bindung eingetragen. Im zweiten Fall wird die zweite Komponente des Eintrags für die Patternvariable ergänzt.

20.4 Anwendung des Mustervergleichs

Im Jahre 1965 wurde der Öffentlichkeit ein Programm zur Simulation von natürlich-sprachlichen Dialogen präsentiert. Dieses Programm, genannt ELIZA, konnte in bestimmten Themenbereichen so geschickt Dialoge führen, daß manche menschlichen Gesprächspartner überzeugt waren, sich mit einem einfühlsamen Menschen unterhalten zu haben. Dabei stand ihnen nur ein Programm gegenüber. Dieses Programm hatte die Sätze (durch einfachen Mustervergleich) daraufhin analysiert, ob es bestimmte Begriffe und Wortkonstellationen enthielt, und dann dazu nach vorgegebenen Regeln Antworten konstruiert. Eine der bekanntesten Anwendungen bestand in der "Führung" von psychotherapeutischen Gesprächen.

Wir wollen hier eine einfache Version eines Programms ELIZA entwerfen, das ein psychotherapeutisches Gespräch karikiert.

Wir legen zuerst die Gesprächsmuster fest, nach denen Eliza arbeitet. Dazu ordnen wir dem Symbol SCHEMA eine Liste von Musterpaaren für Frage und Antwort zu. Jedes Musterpaar ist eine Liste mit zwei Elementen; jedes Element ist ein Muster in Gestalt einer Liste.

```
(SETQ SCHEMA
  '(((ICH BIN +ETWAS ! )
       (WARUM BIST DU +ETWAS ?))
  ((+ MUTTER + ! )
         (ERZÄHLE MIR MEHR ÜBER DEINE FAMILIE !))
  ((ICH KÖNNTE +ETWAS ! )
       (WESHALB ZÖGERST DU ?))
  ((JA + !)
         (ICH VERSTEHE  WAS SAGEN DEINE ELTERN DAZU ? ))
  ((+ HILFE ! )
       (HELFEN KANN ICH AM BESTEN
             ICH BRAUCHE NUR VERTRAUEN ! ))
```

```
((MEIN +V !)
      (WAS HAT DEIN +V NOCH FÜR DICH GETAN ? ))
((ER HAT MIR  +L =ER ! )
      (ES ZEIGT SEIN INTERESSE AN DIR DASS ER DIR
       +L =ER  HAT
        WAS ERHOFFST DU DIR NOCH ? ))
((DIE FRAUEN SIND + ! )
      (IN WELCHER HINSICHT ? ))
((+ NÜTZEN + AUS ! )
      (HAT DAS FÜR DICH GROSSE BEDEUTUNG  ? )))
```

Jedesmal, wenn ELIZA eine Eingabe liest, sucht sie in ihrem SCHEMA nach einem Musterpaar, dessen Eingabemuster zur aktuellen Eingabe paßt.

So ordnet ELIZA der Eingabe

```
      ICH BIN IMMER SO TRAURIG    !
```

das Musterpaar

```
((ICH BIN +ETWAS)
 (WARUM BIST DU +ETWAS))
```

zu. Dabei wird die Mustervariable +ETWAS an die Liste (IMMER SO TRAURIG) gebunden. Mit dieser Bindung wird aus dem zugehörigen Ausgabemuster

```
(WARUM BIST DU +ETWAS ? )
```

die Ausgabe

```
WARUM BIST DU IMMER SO TRAURIG ?
```

konstruiert.

Die Aufgabe, einen Satz zu lesen und diesen in eine Liste umzuwandeln übernimmt die Funktion **READ-SENTENCE**, die wie folgt definiert ist:

```
(DEFUN  READ-SENTENCE  ( )                    ;  leere Parameterliste
  (DO ((SATZ NIL (APPEND SATZ WORT))
       (WORT (READ) (READ)))
      ((MEMBER WORT '(? !)) SATZ)
      (SETQ WORT (LIST WORT))))
```

Die Iterationsvariablen **SATZ** und **WORT** der DO-Iteration im Rumpf von READ-SENTENCE haben als Wert die Liste der bis dahin gelesenen Symbole mit Ausnahme des letzten bzw. das zuletzt gelesene Symbol.
In jedem Iterationsschritt wird festgestellt, ob das zuletzt gelesene Symbol ein Frage- oder Ausrufezeichen ist. Wenn ja, wird die Iteration abgeschlossen und als Wert der Wert von Satz übergeben. Anderenfalls wird der Rumpf der DO-Form abgearbeitet und der nächste Iterationsschritt mit neuen Werten für die Iterationsvariablen vorbereitet.

Jede Eingabe muß durch ein Fragezeichen oder Ausrufezeichen abgeschlossen werden; sobald eines dieser Zeichen gelesen ist, bricht die DO-Iteration ab.

Wir definieren jetzt die Funktion ELIZA selbst

```
(DEFUN ELIZA (DOLLAR SCHEMA)
  (FORMAT T "ICH BIN ELIZA !  WAS BEDRÜCKT  DICH ?")
  (DOTIMES (VAR DOLLAR)
      (LET ((E (READ SENTENCE)))
        (DO ((SCH SCHEMA (CDR SCH))
             (ASS NIL NIL))
            ((SETQ ASS (MATCH  (CAAR SCH)  E NIL))
                       (ANTWORT (CADAR SCH) ASS))
            (IF (NULL  SCH)
                (RETURN-FROM  ELIZA
                    (FORMAT T "ICH MUSS ÜBERRASCHEND
                       WEG ! ÜBER  IHR PROBLEM
                       WERDE ICH NACHDENKEN. KOMMEN
                       SIE DOCH MORGEN WIEDER !"  ))))))
  (FORMAT T "UNSERE  ZEIT IST LEIDER ZU ENDE !
                  AUF WIEDERSEHEN !"))
```

Eliza hat die Parameter DOLLAR und SCHEMA.
Diese legen fest wieviele Fragen beantwortet werden (jede Antwort kostet einen Dollar) und welches Schema dabei zugrundegelegt wird. Wenn ELIZA aufgerufen wird, dann stellt sie sich vor, fragt mitfühlend und geht dann in die DOTIMES-Iteration, durch die genau soviele Fragen beantwortet werden, wie bezahlt wurden.

Die Beantwortung einer Frage erfolgt durch Abarbeiten der LET-Form. Hier wird die Frage durch

die lokale Variable E übernommen. Dann wird in der untergeordneten DO-Schleife versucht, mit Hilfe von SCHEMA (sozusagen nach "Schema-eff") eine Antwort zu konstruieren. Diese DO-Schleife hat die Iterationsvariablen SCH und ASS. SCH hat als Wert den jeweils noch zu prüfenden Rest von SCHEMA; ASS dient als lokale Variable für die aus dem Mustervergleich einer Frage mit einem Fragemuster resultierende A-Liste.

Die SETQ-Form fungiert als Abbruchtest; wenn MATCH beim Vergleich der Frage mit dem Fragemuster (CAAR SCH) ein Resultat ungleich NIL liefert, wird damit die Antwort anhand des Antwortmusters (CADAR SCH) konstruiert und ausgegeben. Damit ist die Beantwortung einer Frage abgeschlossen; die untergeordnete DO-Schleife wird verlassen. Die Funktion ANWORT wird anschließend definiert und beschrieben. Wird das SCHEMA abgearbeitet, ohne daß eine Antwort konstruiert werden kann, so stiehlt sich ELIZA mit einer Ausrede davon. Nach regulärer Abarbeitung der bezahlten Fragen bricht ELIZA den Dialog höflich aber bestimmt ab.

Jetzt zur Funktion ANTWORT; als Argumente braucht sie ein Antwortmuster und eine dazu passende A-Liste. Sie konstruiert die Antwort zunächst als Liste und entfernt beim Ausdrucken mit PRINT-Satz alle inneren und äußeren Klammern.

Die Funktion ANTWORT ist wie folgt definiert:

```
(DEFUN ANTWORT (SCHABLONE ASS)
  (PRINT-SATZ
      (MAPCAN #'(LAMBDA (KEY)
          (COND ( (S-PATTVAR-P KEY)
                  (CDR (ASSOC KEY ASS)))
                ( (M-PATTVAR-P KEY)
                  (CDR (ASSOC KEY ASS)))
                (T (LIST KEY))))
              SCHABLONE)))

(DEFUN PRINT-SATZ (LISTE)
  (TERPRI)
  (MAPCAR #'(LAMBDA (WORT) (PRINC WORT) (PRINC " "))
        LISTE))
```

20.5 Ein Dialog

Der folgende Dialog zeigt die psychotherapeutischen Fähigkeiten von ELIZA.

```
-->(ELIZA 7 SCHEMA)
ICH BIN ELIZA ! WAS BEDRÜCKT DICH ?

  ICH BIN SO TRAURIG !
WARUM BIST DU SO TRAURIG ?

  DIE FRAUEN SIND ALLE GLEICH !
IN WELCHER HINSICHT ?

  SIE NÜTZEN MICH IMMER SO AUS !
HAT DAS FÜR DICH GROSSE BEDEUTUNG ?

  JA DAS RUINIERT MICH FINANZIELL !
ICH VERSTEHE       WAS SAGEN DEINE ELTERN DAZU ?

  MEIN VATER HAT GETOBT !
WAS HAT DEIN VATER NOCH FÜR DICH GETAN ?

  ER HAT MIR EIN NEUES AUTO GESCHENKT !
ES ZEIGT SEIN INTERESSE AN DIR DASS ER DIR EIN
NEUES AUTO GESCHENKT HAT
WAS ERHOFFST DU DIR NOCH  ?

  ICH BRAUCH HILFE  !
HELFEN KANN ICH AM BESTEN ICH BRAUCHE NUR VERTRAUEN      !

  DAS HABE ICH !
UNSERE ZEIT IST JETZT LEIDER ZU ENDE AUF WIEDERSEHN !
UNSERE ZEIT IST JETZT LEIDER ZU ENDE AUF WIEDERSEHN !
```

20.6 Zusammenfassung

Mustervergleich oder Pattern-Matching ist eine wichtige Grundaufgabe der Künstlichen Intelligenz. Es geht darum, für ein gegebenes Muster und eine Datenstruktur festzustellen, ob die Datenstruktur dem Muster entsprechend aufgebaut ist. Im Erfolgsfalle wird häufig Information über die gefundenen Beziehungen zwischen Datenstruktur und Muster benötigt.

Es wurden Funktionen zur Durchführung des Mustervergleichs entwickelt und als Anwendungsbeispiel die Führung von Dialogen durch einen Computer behandelt. Dieses Anwendungsbeispiel erregte seinerzeit viel Aufsehen und ließ gerade bei Laien den Glauben an unbegrenzte Möglichkeiten der Künstlichen Intelligenz entstehen. Wir sehen, daß hier im wesentlichen ein geschickter Mustervergleich und Antworten nach "Schema-eff" zugrundeliegen.

KAPITEL 21:

Datentypen und Datenstrukturen

Wir haben uns bisher vor allem darauf konzentriert, LISP mit seinen charakteristischen Eigenschaften als funktionale Sprache einzuführen und einige typische Anwendungen aufzuzeigen.

LISP ist entscheidend geprägt durch seine grundlegende Datenstruktur, die Listen und deren Verwendung zur einheitlichen Darstellung von Daten und Programmen.

Da LISP-Programme nicht nur durch Listen dargestellt sondern auch einander wie Listen verarbeiten können, verschwindet der Unterschied zwischen Daten und Programmen. Dies ist der Grund für seine hohe Flexibilität und Faszination.

Das heutige LISP kennt neben dem Datentyp Liste noch eine Vielzahl von Standard-Datentypen und vielfältige Möglichkeiten zur Definition weiterer Datentypen.

Dieses Kapitel gibt einen Überblick über die wichtigsten Datentypen in COMMON LISP und führt noch einen wichtigen Datentyp, nämlich "DEFSTRUCT-Strukturen" ein. Schließlich wird noch auf das Konzept der Vielfach-Werte (multiple values) eingegangen. Es ermöglicht, daß eine Funktion auch mehrere Werte übergibt, ohne daß diese in einer Liste, einem Array oder einer anderen Datenstruktur zusammengefaßt sein müssen.

21.1 Datentypen in COMMON LISP

COMMON LISP sieht eine Vielzahl von Datentypen vor. Dabei ist jeder Datentyp eine Menge von LISP-Objekten. Wir sagen, ein Datenobjekt gehört einem bestimmten Typ an, wenn es in der zugehörigen Menge enthalten ist.
Die Datentypen sind bezüglich Mengeninklusion hierarchisch angeordnet.

Auf die wichtigsten Datentypen und zugehörige Typbezeichner gehen wir in den folgenden Abschnitten ein.

21.1.1 Die wichtigsten Datentypen

Zu den wichtigsten Klassen von LISP-Objekten gehören Zahlen, Characters, Symbole, Listen, Arrays, Strukturen und Funktionen. Sie bilden Datentypen, zum Teil mit verschiedenen Untertypen. Bis auf Strukturen - sie werden im nächsten Abschnitt dieses Kapitels eingeführt - haben wir sie bereits kennengelernt, wenn auch nicht mit allen technischen Details. Auf einige dieser Details wird im folgenden hingewiesen.

Zahlen sind in verschiedenen Formen und Darstellungsweisen vorgesehen.

Der Datentyp NUMBER hat in COMMON LISP die Untertypen

- *INTEGER* für ganze Zahlen und zwar ohne Größenbeschränkung, mit Untertypen FIXNUM und BIGNUM (implementationsabhängig), wobei neben Dezimaldarstellung auch Darstellung in beliebiger Basis von 2 bis 36 möglich sein muß.

- *RATIONAL* für rationale Zahlen einschließlich ganzer Zahlen in Quotiendarstellung bzw. ganzzahliger Darstellung; Beispiele sind
  -3/7 , 4/9, 10/5 , 2, 7 usw.

- *FLOAT* für Gleitpunktzahlen mit (implementationsabhängigen) Untertypen
  SHORT-FLOAT, SINGLE-FLOAT,
  DOUBLE-FLOAT und LONGFLOAT,
  wobei in der Schreibweise nur Dezimalzahlen mit oder ohne zusätzliche Zehnerpotenzen zugelassen sind.

- *COMPLEX* für komplexe Zahlen mit einer Darstellung der Gestalt
  #C (<Realteil> <Imaginarteil>) ;
  Untertypen entstehen durch Typfestlegung für Realteil und Imaginarteil.
  Ist dieser Typ gleich RATIONAL, so wird im Falle "Imaginarteil gleich 0" automatisch in die Darstellung als rationale Zahl konvertiert.

Characters sind Datenobjekte vom Typ CHARACTER; ihre Niederschrift besteht aus dem Präfix #\ gefolgt von Character selbst.

Beispiele:
#\g oder #\G oder #\Space
Wichtige Untertypen sind STANDARD-CHAR und STRING-CHAR.

Standard-Characters sind die Characters
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ;
< = > ? @ A B C D E F G H I J K L M N O P Q R S
T U V W X Y Z [\ ] ^ _ ` a b c d e f g h i j k l m n
o p q r s t u v w x y z { : } ~

sowie #\SPACE
(das sind jetzt alle 95 ASCII Standard-Characters) und
#\NEWLINE (für Übergang auf neue Zeile)

Die ersten 94 Characters sind "printing characters", die beiden letzten nicht, deshalb haben sie Namen.

Characters, die über die Standard-Characters hinaus von einer Implementation zur Verfügung gestellt werden, heißen Nicht-Standard-Characters

Beispiele für häufig vorhandene Nicht-Standard-Characters sind
#\π oder #\α oder #\ESCAPE.

Die Verwendung von Nicht-Standard-Characters erschwert die Portabilität von Programmen.

Symbole und Listen
standen von Anfang an im Zentrum unserer Aufmerksamkeit und wurden bereits ausführlich behandelt.

Arrays
gehören zum Datentyp ARRAY. Durch Festlegung von Array-Dimensionen und Typ der Komponenten entstehen viele Untertypen. Dazu gehören insbesondere auch die Untertypen VECTOR und STRING.

Strukturen

kennen wir noch nicht; sie entsprechen den Records in anderen Sprachen und werden im nächsten Abschnitt dieses Kapitels noch ausführlich behandelt.

Funktionen

nennt man in LISP alle Datenobjekte, die zusammen mit Argumenten an FUNCALL oder APPLY zur Auswertung übergeben werden können.

Dazu gehören

- Symbole, deren Funktionszelle aufgrund einer DEFUN-Auswertung entsprechende Einträge enthält
- Compilierte Funktionen
- Lambda-Ausdrücke
- Resultate der Auswertung der speziellen Form FUNCTION bzw. der Funktion SYMBOL-FUNCTION

21.1.2 Typbezeichner

Zur Bezeichnung von Typen dienen Typbezeichner. Einige haben wir bereits kennengelernt, z.B. die Symbole NUMBER, SYMBOL, INTEGER, ARRAY, VECTOR, LIST.

Für kompliziertere Datentypen, die Kombinationen oder Spezialisierungen von einfacheren Typen darstellen, dienen Listen als Typbezeichner.

Beispiele sind

(VECTOR DOUBLE-FLOAT 100)
für Vektoren mit 100 Komponenten des Typs DOUBLE-FLOAT

(VECTOR DOUBLE-FLOAT *) oder gleichbedeutend (VECTOR DOUBLE-FLOAT)
für Vektoren mit beliebig vielen Komponenten des Typs DOUBLE-FLOAT

(ARRAY INTEGER (* * *))
für 3-dimensionales ARRAY, mit Komponenten vom Typ INTEGER

(AND <Typ-1> <Typ-2>...)
für Objekte, die sowohl vom Typ-1 wie vom Typ-2 sein müssen

(INTEGER 5 17)
für $\{x \mid x \text{ ganz}, 5 \leq x \leq 17\}$

(INTEGER (5) 17)
für $\{x \mid x \text{ ganz}, 5 < x \leq 17\}$

(INTEGER 5 *)
für $\{x \mid x \text{ ganz}, 5 \leq x\}$

Mit DEFTYPE können Symbole als abkürzende Typbezeichnungen für komplizierte Typbezeichnungen eingeführt werden.

Mit COERCE kann ein Objekt in ein äquivalentes Objekt eines anderen Typs überführt werden.

Wir gehen darauf nicht mehr näher ein, da es den Rahmen dieser Einführung sprengt.

Die **Zugehörigkeit** eines gegebenen **Objekts** zu einem gegebenen **Typ** kann mit dem Prädikat TYPEP durch einen Aufruf der Gestalt

(TYPEP <Objekt> <Typbezeichner>

geprüft werden.

Für ein gegebenes Datenobjekt kann mit Hilfe der Funktion TYPE-OF ein Datentyp ermittelt werden, dem das Objekt angehört. Dabei liefert der Aufruf
(TYPE-OF <Objekt>)
stets einen Datentyp, auch wenn das Objekt mehreren Datentypen angehört.

21.2 Strukturierte Objekte

Wir kennen bisher i.w. zwei Konzepte zur Bildung strukturierter Objekte nämlich Listen und Arrays.

COMMON LISP bietet darüber hinaus die Möglichkeit, strukturierte Objekte nach Art der "Records" einzuführen. Zur Definition solcher Strukturen dient das Makro DEFSTRUCT. Wir sprechen deshalb auch vom DEFSTRUCT-Konzept und von DEFSTRUCT-Strukturen. Bemerkenswert ist vor allem, daß das LISP-System zu jedem benutzerdefinierten

DEFSTRUCT-Typ automatisch speziell angepaßte Funktionen für Erzeugung, Zugriff und Wertzuweisung definiert und zur Verfügung stellt.

Wozu aber brauchen wir eigentlich diese DEFSTRUCT-Strukturen ?

Betrachten wir die Aufgabe eine Raumfähre zu überwachen. Als Daten interessieren uns auf jeden Fall die Position mit 3 Koordinaten, sagen wir x,y, z und die Geschwindigkeit mit 3 Komponenten für die Änderungsgeschwindigkeiten der Positionskoordinaten und ihre Masse. Der Zustand der Raumfähre wird somit durch 7 Koordinaten beschrieben, die x-Position, x-Geschwindigkeit, y-Position, y-Geschwindigkeit usw bis zur Masse. Faßt man diese 7 Koordinaten in einer Liste zusammen, so kann auf die Koordinaten nur mit CAR, CADR usw. zugegriffen werden. Verwendet man ein Array, so muß über Indizes zugegriffen werden. In beiden Fällen hat man das Problem, daß die Bedeutung von Zugriffsfunktionen wie
(CADDDR COLUMBIA) oder (AREF COLUMBIA 3)
schwer im Gedächtnis zu behalten sind, und damit die Fehleranfälligkeit stark erhöht ist.
Zweckmäßiger wäre es, die Komponenten über Namen anzusprechen, so daß Zugriffe der Gestalt

(SHUTTLE - Z-SPEED COLUMBIA)

möglich sind.

Mit dem DEFSTRUCT-Konzept wird dies in der Tat möglich.

Das Makro DEFSTRUCT definiert eine Struktur. Für unser Beispiel könnte die Struktur definiert werden durch

```
(DEFSTRUCT SHUTTLE
    X-POSITION
    Y-POSITION
    Z-POSITION
    X-SPEED
    Y-SPEED
    Z-SPEED
    MASS )
```

Diese DEFSTRUCT-Form legt fest, daß jedes SHUTTLE ein Objekt mit 7 Komponenten ist. Ihre Auswertung hat folgende Effekte:

- Das Symbol SHUTTLE wird Name eines **Datentyps**. Insbesondere kann dann in **Typprüfungen** der Gestalt
  (TYPEP X 'SHUTTLE)
  festgestellt werden, ob X ein Objekt vom Typ SHUTTLE ist.

- **Ein Prädikat** SHUTTLE-P wird definiert; falls sein Argument ein Objekt vom Typ SHUTTLE ist, liefert es T, andernfalls NIL.

- Eine **"Konstruktorfunktion"** MAKE-SHUTTLE wird definiert, die bei jedem Aufruf eine Datenstruktur mit 7 Komponenten erzeugt. Beispielsweise wird durch
  (SETQ COLUMBIA (MAKE-SHUTTLE))
  dem Symbol COLUMBIA als Wert ein Objekt vom Typ SHUTTLE zugewiesen, deren Komponenten noch keine definierten Werte haben. Anfangswerte für die Komponenten können mit Schlüsselwortargumenten festgelegt werden. So bewirkt

```
(SETQ COLUMBIA (MAKE-SHUTTLE    :X-POSITION 0
                                :Y-POSITION 0
                                :Z-POSITION 0
                                :X-SPEED    0
                                :Y-SPEED    0
                                :Z-SPEED    0
                                :MASS       110))
```

  daß ein neues Objekt vom Typ SHUTTLE erzeugt wird, wobei den durch Schlüsselworte bezeichneten Komponenten die Werte der zugehörigen Argumente zugeordnet werden.
  Wert der SETQ-Form ist dieses neue Objekt; es hat die externe Darstellung

```
#S (SHUTTLE  X-POSITION 0  Y-POSITION 0  Z-POSITION 0
             X-SPEED 0     Y-SPEED 0     Z-SPEED 0
             MASS 110 )
```

  Das Präfix #S zeigt, daß hier eine mit DEFSTRUCT definierte "Struktur" ausgedruckt wird.

- Für die Komponenten werden **Zugriffsfunktionen** definiert, deren Namen aus der Strukturbezeichnung und den Komponentenbezeichnungen gebildet werden.
  Sie lauten hier

SHUTTLE-X-POSITION
.
.
.
SHUTTLE-X-SPEED
.
.
.
SHUTTLE-MASS

Damit gilt zum Beispiel

```
-->(SHUTTLE-MASS COLUMBIA)
110
```

Mit SETF und den Zugriffsfunktionen können auch einzelne Strukturkomponenten geändert werden.

Beispiel:

```
-->(SETF (SHUTTLE-MASS COLUMBIA) 200))
200
```

- Eine **Kopierfunktion** COPY-SHUTTLE wird definiert zum Kopieren von Strukturen des Typs SHUTTLE.

21.3 Mehr zum DEFSTRUCT-Konzept

DEFSTRUCT bietet wesentlich mehr Möglichkeiten, als oben beschrieben. Die gebräuchlichsten werden im folgenden behandelt. Dazu betrachten wir das
Aufrufmuster :

```
(DEFSTRUCT   <Name>
             <Beschreibung-Komponente-1>
                  .
                  .
                  .
             <Beschreibung-Komponente-n>)
```

Das erste Argument muß ein Symbol sein; es wird Name des neu definierten Datentyps, dem alle Inkarnationen dieser Struktur angehören. Die Funktion TYPEP akzeptiert ihn als Typbezeichner. Dieser Name wird auch als Wert der DESTRUCT-Form übergeben.

Die Beschreibung einer Komponente besteht im einfachsten Fall nur aus einem Symbol; diesen Fall haben wir am Beispiel des Typs SHUTTLE kennengelernt. In diesem Fall veranlaßte ein Aufruf von

MAKE-<Typname>

die Anlage einer Struktur, in der nur diejenigen Komponenten definierte Werte erhalten, die im Aufruf der Konstruktorfunktion MAKE-<Typname> durch Schlüsselwortargumente vertreten sind.

Beispiel:

Der Aufruf

```
-->(SETQ PLUTO (MAKE-SHUTTLE :MASS 250))
#S(SHUTTLE   X-POSITION NIL  Y-POSITION NIL  Z-POSITION NIL
             X-SPEED NIL     Y-SPEED NIL     Z-SPEED NIL
             MASS 250)
```

bewirkt die Konstruktion eines neuen Datenobjekts mit Namen PLUTO vom Typ Shuttle, von dessen 7 Komponenten nur MASS einen definierten Wert erhält.

Es ist jedoch möglich, für die Komponenten auch Default-Anfangswerte festzulegen. Dazu dienen Komponentenbeschreibungen, die nicht nur aus dem Komponentennamen bestehen, sondern aus einer zweielementigen Liste der Gestalt

(<Komponentenname> <Default-Anfang>)

mit dem Namen der Komponente und einer Form für den Default-Anfangswert.

Wenn jetzt ein Aufruf der Konstruktorfunktion MAKE-<Typname> für eine Komponente kein Schlüsselwort-Argument enthält und damit keinen Anfangswert festlegt, wird automatisch der Wert der zugehörigen Form Default-Anfang aus der Komponentenbeschreibung der DEFSTRUCT-Form ermittelt und der Komponente als Anfangswert zugeordnet.
Der Wert von Default-Anfang wird jedoch nicht im Zuge der Auswertung der DEFSTRUCT-Form ermittelt, sondern grundsätzlich erst bei Auswertung der Konstruktorfunktion MAKE-<Typname> , allerdings unter Zugrundelegung der lexikalischen Umgebung, welche die

DEFSTRUCT-Form enthält. Dieselbe Form kann dabei in verschiedenen MAKE-Aufrufen durchaus zu verschiedenen Anfangswerten führen.

Beispiel:
Die Gehälter von Beschäftigten werden oft recht geheimnisvoll behandelt. Wir berechnen im folgenden Anfangsgehälter durch Auswerten ein- und derselben Form (* FACTOR 1000) und stellen trotzdem Unterschiede fest:

```
-->(DEFSTRUCT EMPLOYEE NAME (SALARY (* FACTOR 1000)))
EMPLOYEE

-->(SETQ FACTOR 2)
2

-->(LET  ((FACTOR 3))
        (SETQ Y  (MAKE-EMPLOYEE
                          :NAME JOHNSON))
        (SETQ X  (MAKE-EMPLOYEE
                          :NAME CARTER
                          :SALARY (* FACTOR 1000))))
#S (EMPLOYEE NAME CARTER SALARY 3000)

-->(EMPLOYEE-SALARY X)                          ;  Auswertung in der durch LET
3000                                            ;  eingeführten Umgebung
                                                ;  bringt mehr als ...

-->(EMPLOYEE-SALARY Y)                          ;  Auswertung in der lexika-
2000                                            ;  lischen Umgebung der
                                                ;  DEFSTRUCT-Form

-->(SETQ FACTOR 5)                              ;  Hochsetzen des Faktors für
5                                               ;  ...

-->(SETQ Z (MAKE-EMPLOYEE :NAME REAGAN))        ;  Auswertung in der lexika-
#S  (EMPLOYEE NAME REAGAN SALARY 5000)          ;  lischen Umgebung der
                                                ;  DEFSTRUCT-Form
```

21.5 Vielfach-Werte (Multiple Values)

Wir sind bisher immer davon ausgegangen, daß die Auswertung einer LISP-Funktion stets genau ein LISP-Objekt als Wert liefert. Dieses LISP-Objekt konnte aber durchaus die Gestalt einer Liste, eines Arrays oder einer DEFSTRUCT-Struktur haben, also aus Komponenten zusammengesetzt sein.

Wenn eine Funktion mehrere Objekte berechnen sollte, mußten wir diese bisher stets zu einem umfassenden Objekt bündeln, das dann als Wert übergeben wurde.

Gelegentlich kann es wünschenswert sein, verschiedene Objekte, die von einer Funktion berechnet werden, ungebündelt als Wert übergeben zu können. So mag manchmal die Struktur eines Programms klarer erscheinen, wenn verschiedene Objekte nicht mit Gewalt zusammengefaßt werden müssen.Weiter kann die Zusammenfassung, etwa in Listen, aufgrund der vielen hierfür nötigen CONS-Zellen sehr speicheraufwendig werden.

COMMON LISP stellt deshalb einen Mechanismus bereit, mit dem auch "ungebündelte" Werte - wir sprechen von **Vielfach-Werten** - erzeugt und übergeben werden können. Sowohl zur Erzeugung wie zur Übernahme von Vielfachwerten benötigt man spezielle Formen. Wenn eine Funktionsauswertung Vielfach-Werte liefert, aber die aufrufende Funktion nicht ausdrücklich Vielfach-Werte fordert, dann wird an die aufrufende Funktion nur der erste Wert übergeben, alle weiteren Werte werden ignoriert. Wenn eine Funktionsauswertung keinen Wert liefert, dann erhält die aufrufende Funktion den Wert NIL.

Die wichtigste Systemfunktion zur Erzeugung von Vielfach-Werten ist VALUES. Sie kann beliebig viele Argumente übernehmen und übergibt deren Werte als Vielfach-Werte.

Aufrufmuster:
(VALUES <Arg-1> ... <Arg-n>) mit $n \geq 0$

Auswertung:
Die Argumente Arg-1 bis Arg-n werden der Reihe nach ausgewertet. Die resultierenden Werte werden in der Reihenfolge der Argumente als Vielfach-Werte übergeben.

Ist die letzte Form im Rumpf einer Funktion etwa ein Aufruf von VALUES mit vier Argumenten, dann liefert diese Funktion stets die vier Werte dieser Argumente.

Der Ausdruck (VALUES) übergibt null Werte. Dies ist das Standardbeispiel für einen Funktionsauruf, der keine Werte übergibt.
Durch Aufruf von

(APPLY #'VALUES <-Liste->

kann man offensichtlich auch die Elemente einer Liste als Vielfach-Werte übergeben. Dasselbe erreicht man mit der Systemfunktion VALUES-LIST durch einen Aufruf der Gestalt

(VALUES-LIST <-Liste->)

für ein Argument, dessen Wert eine Liste ist.

Wir erwähnen **noch einige Systemfunktionen**, die Vielfach-Werte liefern.

Ein **Beispiel** ist die Funktion FLOOR; sie hat das **Aufrufmuster:**

(FLOOR <-Zahl-> <-Divisor->)

Das erste Argument ist obligatorisch; sein Wert muß eine Zahl sein. Das zweite Argument ist optional; sein Wert muß eine Zahl ungleich Null sein; Default-Wert ist die Zahl 1.

Werte:
Erster Wert ist die größte ganze Zahl, die nicht größer ist als der Quotient
"Zahl durch Divisor" (das gab den Namen "floor" = Boden)
Zweiter Wert ist die Differenz "<Zahl> - <erster Wert> × <Divisor>"

Neben FLOOR gibt es noch weitere "Rundungsfunktionen", die Vielfachwerte liefern, nämlich CEILING , TRUNCATE und ROUND.

Sie haben Aufrufmuster wie FLOOR und werden fast genauso ausgewertet; es wird nur zum Erhalt des ersten Wertes das Ergebnis der Division anders "gerundet", und zwar

- bei CEILING zur kleinsten ganzen Zahl nicht kleiner als das erste Argument (das gab den Namen "ceiling")
- bei TRUNCATE zur nächsten ganzen Zahl, deren Absolutbetrag nicht größer ist als der Absolutbetrag des Quotienten
- bei ROUND zur nächsten ganzen Zahl, im Falle der Äquidistanz von zwei ganzen Zahlen zur geraden Zahl.

Obwohl jetzt z.B. FLOOR zwei Werte liefert, gilt

```
-->(PRINT (FLOOR -17 4))          ; -4.25 wird durch FLOOR auf -5 "gerundet"
-5
```

Dies bestätigt, daß eine aufrufende Funktion, die nicht ausdrücklich Vielfach-Werte fordert, auch keine bekommt; sie erhält nur den ersten Wert.

21.5 Zur Übernahme von Vielfach-Werten

Zur Übernahme und Verarbeitung von Vielfach-Werten benötigt man eigene Funktionen. Die wichtigste ist die spezielle Form MULTIPLE-VALUE-LIST
mit dem **Aufrufmuster:**

(MULTIPLE-VALUE-LIST <Form>)

Auswertung:
Die bei Auswertung der Form resultierenden Vielfach-Werte werden in einer Liste zusammengefaßt, die als Wert des MULTIPLE-VALUE-LIST-Aufrufs übergeben wird.

Beispiel:

```
-->(MULTIPLE-VALUE-LIST (FLOOR -5 2))
(-3 1)
```

MULTIPLE-VALUE-LIST ist offensichtlich invers zur oben erwähnten Funktion VALUES-LIST.

Wir erwähnen noch weitere Systemfunktionen zur Übernahme und Verarbeitung von Vielfach-Werten.

Die Funktion MULTIPLE-VALUE-CALL ist eine Verallgemeinerung von FUNCALL; sie hat das **Aufrufmuster:**

(MULTIPLE-VALUE-CALL <Funktionales Objekt> <Form-1> ... <Form-n>)

Auswertung:
Zuerst werden das funktionale Objekt sowie Form-1 bis Form n ausgewertet. Dann werden die von Form-1 bis Form-n gelieferten Vielfach-Werte bzw. Werte als Argumentwerte zur Auswertung der vom funktionalen Objekt dargestellten Funktion benutzt. Der dabei resultierende Wert wird Wert des MULTIPLE-VALUE-CALL-Aufrufs.

Beispiel:
```
-->(MULTIPLE-VALUE-CALL #'+ (FLOOR 6 4) (FLOOR 19 5))
10
```
Die erste FLOOR-Form liefert die Werte 1 und 2, die zweite FLOOR-Form liefert die Werte 3 und 4 ; alle werden Argumente von + .

Das Makro MULTIPLE-VALUE-BIND ist eine Modifikation von LET; es hat das **Aufrufmuster:**

(MULTIPLE-VALUE-BIND (<Variablen-Liste> <Values-Form> <Rumpf>)

Das erste Argument ist eine Liste von Variablen;
das zweite eine Form die Vielfach-Werte liefert; darauf folgt ein Rumpf mit beliebig vielen Formen.

Auswertung:
Zuerst wird die Values-Form ausgewertet; an die von ihr gelieferten Werte werden die Variablen der Reihe nach gebunden. Liefert die Values-Form mehr Werte als Variable vorhanden sind, so werden die überzähligen Werte nicht berücksichtigt; liefert sie weniger Werte, so werden die

unversorgten Variablen an NIL gebunden. Unter diesen Variablenbindungen wird der Rumpf wie bei LET abgearbeitet. Der Wert der letzten Form wird Wert des MULTIPLE-VALUE-BIND-Aufrufs.

Beispiel:

```
-->(MULTIPLE-VALUE-BIND (X Y) (FLOOR 7 3) (+ X Y))
3                              ; X wurde an 2 und Y an 1 gebunden)
```

MULTIPLE-VALUE-SETQ ist eine Verallgemeinerung von SETQ und hat das
Aufrufmuster:

(MULTIPLE-VALUE-SETQ <Variablenliste> <Form>)

Auswertung:
Als **Seiteneffekt** werden die bei Auswertung der Form resultierenden Werte den Variablen der Liste zugewiesen. Liefert die Form mehr Werte als Variable vorhanden sind, so bleiben die überzähligen Werte unberücksichtigt; liefert sie weniger Werte, so wird jeder unversorgten Variablen der Wert NIL zugewiesen. Die MULTIPLE-VALUE-SETQ-Form liefert stets **genau einen Wert**, nämlich den ersten Wert der Form, bzw. NIL, wenn die Form keinen Wert liefert.

Beispiel:

```
-->(MULTIPLE-VALUE-SETQ (X Y Z) (FLOOR 6 4))
1                                   ;  erster Wert von FLOOR
-->Y                                ;  zweiter Wert von FLOOR
2
-->Z                                ;  überzählige Variable
NIL
```

Für die Weitergabe von Vielfachwerten gelten folgende Regeln:

EVAL liefert Vielfachwerte, wenn die auszuwertende Form solche produziert.

APPLY, FUNCALL und MULTIPLE-VALUE-CALL
geben die von der angewandten bzw. gerufenen Funktion gelieferten Vielfachwerte weiter.

PROGN sowie Konstruktionen mit implizitem PROGN geben Vielfach-Werte weiter.

IF gibt Vielfach-Werte seiner "Then- und Else-Formen" weiter.

COND gibt Vielfach-Werte weiter, wenn diese aus der Auswertung einer Klausel resultieren, die nicht nur aus einer Form bestand.

Niemals weitergegeben werden Vielfach-Werte von den Formen
SETQ, MULTIPLE-VALUE-SETQ und PROG1;
doch gibt es ein eigenes MULTIPLE-VALUE-PROG1 zur Weitergabe von Vielfach-Werten.

Um sicherzustellen, daß von einer Form nur ein Wert weitergegeben wird, verwendet man (VALUES <Form>).

Die wichtigste Regel bei Vielfach-Werten aber lautet:

Wenn eine Form als Argument in einem Funktionsaufruf auftritt, so wird stets genau ein Wert, nämlich der erste, verwendet und zwar unabhängig davon, wieviele Werte die Form tatsächlich produziert.

Beispiel:

```
-->(CONS (FLOOR 4 3) '(VALUE AND NOTHING ELSE))
(1 VALUE AND NOTHING ELSE)
```

21.6 Zusammenfassung

COMMON LISP sieht eine Vielzahl von Typen für Datenobjekte vor. Dabei ist die Typeigenschaft ausdrücklich den Datenobjekten, nicht aber den Variablen zugeordnet. Jede Variable kann im Prinzip jedes LISP-Objekt als Wert haben.

Jeder Datentyp ist eine Menge von Datenobjekten. Verschiedene Datentypen sind nicht notwendig disjunkt. Zu jedem Datentyp gibt es einen Typ-Bezeichner, der insbesondere für Typüberprüfungen verwendet wird.

Wir hatten bisher zwei Konzepte für strukturierte Datenobjekte kennengelernt, nämlich Listen und Arrays. Auf ihre Komponenten wird über iterierte Anwendung von CAR und CDR bzw. über Indizes zugegriffen.

Mit DEFSTRUCT können nun auch Strukturen definiert werden, deren Komponenten über Namen angesprochen werden.

Eine Funktion gibt im Normalfall genau einen Wert, der natürlich ein beliebig strukturiertes Objekt sein kann. Gelegentlich ist es wünschenswert, auch mehrere Werte übergeben zu können, ohne diese dazu in einer Liste, einem Array oder einer Struktur zusammenfassen zu müssen. Dies ist möglich mit Hilfe des Konzepts der Vielfach-Werte. Erzeugung und Übernahme solcher Vielfachwerte sind jedoch nur unter ganz besonderen Vorkehrungen möglich. Erzeugt eine Funktion auf diese Weise Vielfach-Werte, so ist einer als erster Wert ausgezeichnet. Tritt ein Aufruf dieser Funktion dann irgendwo als Argument auf, so wird von der aufrufenden Funktion nur dieser erste Wert übernommen, sofern die Übernahme der Vielfach-Werte nicht explizit angeordnet ist.

KAPITEL 22:

Zur Entwicklung von Programmen

Erstes Ziel jeder Art von Programmierung ist die Entwicklung zuverlässiger und fehlerfreier Programme, die von einem größeren Kreis von Benutzern mit Erfolg eingesetzt und dabei wechselnden Anforderungen angepaßt werden können.
Das bedeutet insbesondere, daß die Programme auch Außenstehenden verständlich sind bzw. gemacht werden können.
Damit die Programmentwicklung überhaupt erfolgreich verlaufen kann, muß das Programm sowohl als Ganzes wie in Teilen auf Fehler getestet werden können.
Wenn Fehler entdeckt oder veränderte Anforderungen berücksichtigt werden müssen, muß das Programm modifiziert, also gewartet werden können.

Den genannten Zielen ***Verständlichkeit, Testbarkeit und Wartbarkeit*** sind nun vor allem zwei Arten von Maßnahmen förderlich:

- Auf Seiten des Programmierers
  Ordnung und Disziplin im Programmentwurf

- Auf Seiten des Systems
  Bereitstellung von Hilfsmitteln, die den Programmierer bei seiner Arbeit unterstützen.

Mit diesen Maßnahmen beschäftigt sich dieses Kapitel.

22.1 Strukturiertes Programmieren

Die in diesem Buch vorgestellten Programme waren letztlich alle so einfach und kurz, daß wir uns von ihrer Korrektheit stets leicht überzeugen konnten.

Die in der Praxis eingesetzten Programme sind häufig um Zehnerpotenzen umfangreicher, entsprechend schwieriger zu überschauen und hinsichtlich ihrer Korrektheit nicht mehr so leicht zu beurteilen.

Das Ziel fehlerfreie Programme zu erstellen, die verständlich, wartbar und testbar sind, wird umso leichter erreicht, je übersichtlicher der Programmaufbau, also die Struktur des Programms ist, je verständlicher die einzelnen Programmkomponenten selbst sind und je klarer das Zusammenspiel der Programmkomponenten geregelt ist. Mit anderen Worten, die Programme müssen übersichtlich gegliedert sein in überschaubare Einheiten und Subeinheiten, wir sprechen von Modulen, und der Informationsfluß zwischen diesen Modulen muß durch klar definierte Schnittstellen eindeutig festgelegt sein. Diese Eigenschaften eines strukturierten oder modularen Aufbaus meinen wir letztlich, wenn wir von "strukturiertem Programmieren" sprechen. Sie erleichtern Entwicklung, Test undWartung von Programmen wesentlich.

Den aufgestellten Forderungen nach übersichtlicher Strukturierung und klaren Schnittstellen kommt die Natur der Sprache LISP in besonderer Weise entgegen.

Struktur, Auswertung und Abarbeitung von Programmen sind in erster Linie durch die Aufrufbeziehungen zwischen den das Programm bildenden Funktionen bestimmt. Eine tief gestaffelte Hierarchie von Funktionsaufrufen ist der Normalfall. Funktionen, die bestimmte Aufgaben zu erfüllen haben, reichen Teilaufgaben an untergeordnete Funktionen weiter und übergeben ihrerseits nach Bearbeitung der Aufgabe das Resultat an übergeordnete Funktionen.

LISP ist somit eine Sprache, welche die Funktionsabstraktion, wir könnten auch sagen, die Strukturierung in Funktionseinheiten, in ganz besonderer Weise fördert.

Wir erkennen damit die Bedeutung der folgenden Gesichtspunkte für "gute" LISP-Programme:

- Die zu bearbeitenden Aufgaben sollten soweit möglich und sinnvoll strukturiert also auf Teilaufgaben zurückgeführt werden und durch Aufruf entsprechender zugeordneter Funktionen gelöst werden. Dies führt auf eine hierarchische Strukturierung der Aufrufbeziehungen zwischen Funktionen.
- Jeder Funktionsdefinition sollte eine klare Spezifikation der zu lösenden Aufgabe zugeordnet sein. Der Lösungsweg sollte klar dokumentiert sein.
- Für die Formulierung eines Funktionsaufrufs sollten keine detaillierten Kenntnisse der Arbeitsweise der Funktion und der Organisation ihrer Daten nötig sein.
- Die einzelnen Funktionen sollten "kurz" und "übersichtlich" sein.
- Die einzelnen Funktionen sollten so wenig wie möglich auf spezielle Situationen zugeschnitten sein, in denen sie aufgerufen werden; d.h. sie sollten möglichst allgemein verwendbar sein.

- Die einzelnen Funktionen sollten Meldungen abgeben, wenn Situationen auftreten, die eine korrekte Abarbeitung unmöglich machen.

Diese Prinzipien sollen noch durch einige konkrete Hinweise ergänzt werden:

Für Funktionen und Variable sollten grundsätzlich Namen gewählt werden, die ihre Bedeutung erläutern; durch Zusammensetzung von Namen mit Bindestrichen (Beispiel: WORST-CASE-ELEMENT) wird dies wesentlich erleichtert.

Die Forderung nach klaren Schnittstellen von Funktionen wird verletzt, wenn globale oder nichtlokale Variable in Funktionen auftreten und dazu benutzt werden, um Werte zwischen Funktionen hin und her zu transportieren. Globale Variable sollten grundsätzlich nur für wirklich globale Daten verwendet werden und wie *STANDARD-INPUT* oder *ERROR-OUTPUT* mit einem Stern am Anfang und Ende als global gekennzeichnet sein.

LISP ist nicht vollständig funktional. Die Verwendung von Seiteneffekten verletzt die Prinzipien der funktionalen Programmierung und ist eine der ergiebigsten Quellen für Irrtümer, zuerst bei der Programmentwicklung und dann bei der Fehlersuche. Am häufigsten wird hier mit SETQ "gesündigt". SETQ's und Seiteneffekte sollten deshalb möglichst sparsam verwendet und sorgfältigst dokumentiert werden.

22.2 Unterstützung durch Programmentwicklungssysteme

Die Kommunikation zwischen Benutzer und LISP-System wird im einfachsten Fall über den Interpretationszyklus, wir sagen auch auf dem "Top-Level" abgewickelt.

Danach liest das System jeweils eine Form ein, veranlaßt deren Auswertung unter den gegebenen Bindungen, gibt den erhaltenen Wert aus und startet diesen Zyklus ("Read-Eval-Print-Schleife") von Neuem.

Bei jedem Schleifendurchlauf kann der Benutzer mit seiner Eingabe die Erzeugung neuer oder die Modifikation vorhandener LISP-Objekte veranlassen und so die Auswertungsumgebung verändern.
Durch Steuerung der Ein- und Ausgabeströme kann die jeweilige Umgebung oder ein Teil davon auf externe Dateien gespeichert werden. Umgekehrt kann auch durch Lesen aus externen Dateien die Umgebung angereichert werden. Wie dies in einem LISP-System durch Aufruf von

System-Funktionen bzw. Makros auf dem Top-Level veranlaßt werden kann, haben wir gesehen. Neben diesen auf dem Top-Level einsetzbaren Funktionen, verfügen heutige LISP-Systeme noch über eine größere Zahl von Werkzeugen, welche die Entwicklung und Ablaufüberwachung von Programmen erleichtern. Sie bilden zusammen ein sogenanntes **Programmentwicklungssystem** (auch **Programmiersystem** oder **Programmierumgebung** genannt). Wir verstehen darunter eine Menge von aufeinander abgestimmten Programmen, die als Werkzeuge bei der Programmentwicklung verwendet werden. Diese Werkzeuge sind also selbst Programme, die andere Programme manipulieren.
Da LISP sowieso keinen grundsätzlichen Unterschied zwischen Daten und Programmen kennt, sind die Werkzeuge dieser Programmentwicklungssysteme im allgemeinen auch selbst in LISP implementiert und verarbeiten die von ihnen bei der Entwicklung überwachten Programme als Daten.
Die Programmentwicklungssysteme sind heute sehr leistungsfähig geworden und unterstützen den Benutzer in allen Phasen der Programmentwicklung, vom Entwurf bis zur endgültigen Implementierung und zur effizienten Programmausführung. Die Entwicklung von Programmen wird dadurch nicht nur wesentlich einfacher und schneller, sondern insbesondere auch weniger fehleranfällig.

Umfangreiche Programme werden, wie wir gesehen haben, aus guten Gründen strukturiert - wir sagen auch - modular aufgebaut. Die einzelnen Module werden häufig von verschiedenen Programmierern erstellt oder stammen aus einer Programmbibliothek. Somit besteht auch bei der Erstellung neuer Programme die Tätigkeit des Programmierers überwiegend in der Integration und Modifikation von Programmmoduln. Gerade hier kommt es besonders auf die Leistungsfähigkeit der Werkzeuge des Programmentwicklungssystems an. Doch ist ihr Gebrauch häufig nicht gerade einfach. Die Einarbeitung in ein Programmentwicklungssystem erfordert häufig sogar mehr Zeit, als das Erlernen der Programmiersprache selbst. Deshalb ist es besonders wichtig, daß das Programmentwicklungssystem benutzerfreundlich aufgebaut ist. Dazu gehört insbesondere, daß die einzelnen Module optimal aufeinander abgestimmt sind, kompatible Schnittstellen haben und in analoger Weise bedient werden können. Dazu gehört weiter, daß der Benutzer jederzeit Informationen darüber erhalten kann, wo er sich gerade befindet, sowie welche Schritte er jeweils ausführen und wie er zusätzliche Hilfestellung erhalten kann.

22.3 Die Komponenten eines Programmentwicklungssystems

Zu den wichtigsten Komponenten eines Programmentwicklungssystems gehören Werkzeuge für

- Entwurf und Modifikation (sog. Editor)
- strukturierte Darstellung und Ausgabe (sog. Pretty-Printer)
- Archivierung und Verwaltung verschiedener Programm- bzw. Modulversionen im File-System
- Spezifikation und Dokumentation
- leichtere Handhabung auf dem Top-Level.
- Überwachung der Programmausführung, Fehlererkennung und Fehlerkorrektur (sog. Debugger)

Die Eigenschaften dieser Werkzeuge sind weitgehend implementationsabhängig. Die Definition von COMMON LISP enthält diesbezüglich nur einige grundsätzliche Aussagen und überläßt die Ausgestaltung der jeweiligen Implementation. Wir gehen im folgenden kurz auf die einzelnen Werkzeuge ein.

Zum Editor

Die Aufgabe eines LISP-Editors besteht darin, den Benutzer beim Schreiben und Editieren von Programmen zu unterstützen und einen übersichtlichen und benutzerfreundlichen Übergang in andere Module zu ermöglichen, z.B. zur Auswertung einzelner Funktionen und Programmstücke, zum Debugging oder zur Überarbeitung der Dokumentation. Wir sagen, die Schnittstelle zu den anderen Komponenten des Systems muß benutzerfreundlich sein.

Jeder Editor stellt zunächst Möglichkeiten zur Niederschrift und Modifikation bzw. Korrektur von Programmstücken bereit.

LISP-Editoren werden danach unterschieden, ob sie "strukturorientiert" (man sagt auch "syntaxorientiert") oder "bildschirmorientiert" arbeiten.

Beim sogenannten strukturorientierten Editor orientiert man sich bei der Eingabe des Programms, beim Navigieren im Programm und bei der Modifikation an der Listenstruktur, also der LISP-Syntax. Das vom Editor zu einem bestimmten Zeitpunkt bearbeitete Teilstück des

Programms, also der gerade zu editierende Term heißt Edit-Form des Programms. Typischerweise verfügt ein "Strukturediter" über Befehle

- zum Wechsel des zu bearbeitenden Bereichs, also der Edit-Form
- zum Suchen in der Edit-Form und zum rekursiven Aufruf des Editors für Teile derselben
- zur Modifikation der Edit-Form
- zum Rückgängigmachen von Modifikationen
- zur Ausgabe der Edit-Form in übersichtlicher ggf. verkürzter Darstellung (mit beschränkter Schachtelungstiefe)
- zur Auswertung von Teilformen durch den Interpreter
- zur Definition neuer Editierkommandos (als Makros für bestimmte Kombinationen vorhandener Editierkommandos)

usw.

Bei strukturorientierten Editoren fehlen typischerweise Kommandos, um auf einzelne Zeichen, Zeilen oder Bildschirmbereiche zuzugreifen, wie dies von Texteditoren her bekannt ist.

Manche LISP-Editoren bauen auf derartigen zeichen-, zeilen-, oder allgemein bildschirmorientierten Editoren auf, verfügen aber dann neben den üblichen Operationen zur Textverarbeitung wie

- Bewegung des Cursors (Zeigers)
- Löschen, Ersetzen, Kopieren von Zeichen, Zeilen oder Bereichen
- Setzen von Marken für schnelle Positionierung des Zeigers
- Fensterverwaltung

über zusätzliche Operationen zur Manipulation von LISP-spezifischen Listenstrukturen, wie

- Cursorbewegungen zur nächsten bzw. vorhergehenden LISP-Form
- Löschen, Ersetzen und Kopieren von LISP-Formen
- Setzen und Löschen von Klammerpaaren
- Suchen der Definition einer Funktion

und natürlich Möglichkeiten zum Übergang in andere Module des LISP-Systems wie

- Auswertung einer LISP-Form
- Compilieren einer LISP-Form
- Definition neuer Editier-Funktionen

Der Pretty-Printer

Die Aufgabe des Pretty-Printers besteht darin, die externe Darstellung von LISP-Objekten so aufzubereiten, daß ihre Struktur erkennbar wird. Die Aufbereitung zu einer gefälligen und leicht durchschaubaren Struktur kann bezüglich verschiedener Kriterien vorgenommen werden. Diese Kriterien können implementationsabhängig sein.

Zum Archivierungssystem

Ein leistungsfähiges Archivierungssystem ist wünschenswert, weil bei der interaktiven Programmentwicklung häufig Programmteile in unterschiedlichen Versionen erstellt und getestet werden. Hier sollte insbesondere eine automatische Buchhaltung der alternativen bzw. durch Änderungen auseinander hervorgegangenen Versionen möglich sein.

Zum Top-Level

Die Interaktion mit dem LISP-System läuft normalerweise über den Top-Level in Form der "Read-Eval-Print-Schleife". Die Definition von COMMON-LISP läßt für den Top-Level ausdrücklich eine Ausgestaltung der Möglichkeiten des Dialogs zu, um so dem Benutzer die Handhabung zu erleichtern.

Besonders wichtig ist das Einholen von Informationen:

- Dies kann die zuletzt zur Auswertung eingegebenen Formen und ihre Werte betreffen: Dazu verfügt LISP über globale Variable, welche diese Formen und ihre Werte aufnehmen. Die wichtigsten sind - , + , ++ , +++ sowie * , ** und ***.
  Der Wert von
  - gibt immer die gerade in Auswertung befindliche Form
  + bzw. ++ bzw. +++ gibt die zuletzt bzw. an vorletzter Stelle bzw. an drittletzter Stelle ausgewertete Form
  * bzw. ** bzw. *** gibt den für + bzw. ++ bzw. +++ ermittelten Wert.

- Falls etwa vergessen wurde, welche Bedeutung ein LISP-Objekt hat, so kann dies mit DESCRIBE ermittelt werden.
  Aufrufmuster:
  (DESCRIBE <Objekt>)

Auswertung:
Als **Seiteneffekt** wird über die Standardausgabe Information über das Objekt ausgegeben. Art und Umfang der Information sind implementationsabhängig. DESCRIBE liefert **keinen Wert**, d.h. es liefert dasselbe, was auch der Ausdruck (VALUES) liefert, nämlich null Werte.

Beispiele:
Antworten auf DESCRIBE sehen typischerweise wie folgt aus:

```
-->(DESCRIBE 'CAAR)
It is a symbol 347DCD
Name:      "CAAR"
Value:     Unbound
Function:  #<Compiled-Function CAAR 264FDF>
PLIST:     NIL
Package:   #<Package "LISP" 385523>

-->(DESCRIBE '-)
It is a symbol 38400D
Name:      "-"
Value:     (DESCRIBE (QUOTE -))
Function:  #<Compiled-Function - 29596 F
PLIST:     NIL
Package:   #<Package "LISP" 385523>
```

Auf jede der beiden Anfragen wird geantwortet, daß ein Symbol vorliegt. Dann wird Information über die Komponenten P-Name, Wert, zugeordnete Funktion, Property-Liste und Home-Package des jeweiligen Symbols ausgegeben.
Die Zahlen geben zugehörige Speicheradressen.
Man beachte die Wertangabe zum Symbol - : Es hat als Wert immer die zuletzt eingegebene Form, hier also die Form (DESCRIBE (QUOTE -)).

- Zu einem Symbol, von dem nur ein Teilstück bekannt ist, können mit APROPOS alle Symbole ermittelt werden, welche das bekannte Teilstück enthalten.

Aufrufmuster:
(APROPOS '<Symbol-Teilstück>)

Auswertung:
Als Seiteneffekt werden über die Standardausgabe alle Symbole, in deren Name das Argument als Teilstück auftritt, mit erläuternden Informationen ausgegeben. APROPOS liefert **keinen Wert.**

Beispiel:
-->(APROPOS 'RPL)
RPLACD, Function
RPLACA, Function

Der Leser wird aufgefordert, die Funktion APROPOS mit Argumenten wie 'ON oder 'SET aufzurufen.

Übrigens hat APROPOS noch ein zweites (optionales) Argument zur Angabe eines Package, wenn sich die Suche auf ein solches beschränken soll.

Wir erwähnen noch Möglichkeiten für

- automatische Fehlerbehandlung einfacher Fehler - insbesondere von Schreibfehlern mit DWIM-Paketen (von Do What I Mean)
- automatische Protokollierung des Dialogs oder eines Teils davon auf einer Datei (mit Hilfe der Funktion DRIBBLE)
- Wiederholung von Auswertungen (mit REDO)
- Rückgängigmachen von Auswertungen (mit UNDO)

Zum Debugger

Programme sind selten auf Anhieb fehlerfrei und leisten oft lange nicht, was sie sollen: Irgendetwas läuft sich tot, eventuell mit einer unverständlichen Fehlermeldung, die nicht wirklich weiterhilft.
Die Korrektur von Fehlern setzt ihre Erkennung voraus; um diese zu erleichtern, stellt der Debugger Werkzeuge zur Überwachung der Programmausführung bereit. Damit sollen möglichst schnell Fehler erkannt und das Programm wieder "zum Laufen" gebracht werden können. Der Benutzter wird so unterstützt, die notwendigen Verbesserungen "on the fly", d.h. im Rahmen der Programmausführung, vorzunehmen.

Der Debugger verfügt im allgemeinen über eine eigene Read-Eval-Print-Schleife, allerdings mit einigen Unterschieden zum Top-Level. Der Debugger versteht wie der Top-Level einige spezielle Kommandos. Fast immer ist darunter auch eines, das als Hilfe für den Benutzer die Ausgabe aller gerade verfügbaren Kommandos veranlaßt.

Zum Eintritt in den Debugger kommt es i.a. dadurch, daß die Auswertung einer Form nicht korrekt abgeschlossen werden konnte. Wenn diese Form selbst im Rahmen der Abarbeitung übergeordneter EVAL-Aufrufe aufgerufen wurde, so haben wir eine ganze Hierarchie übergeordneter geschachtelter Aufrufe, die noch nicht vollständig abgearbeitet sind. Die entsprechende Information ist in einer nach dem Kellerprinzip angelegten Datenstruktur, dem sog. Control-Stack, abgelegt. Die Liste der erst teilweise ausgewerteten Formen auf dem Control-Stack wird als "Backtrace" bezeichnet. Die einzelnen Einträge im Control-Stack heißen Frames. Der oberste Frame gibt diejenige Form, deren Auswertung zum Eintritt in den Debugger führte. Der unterste Frame gibt die auf dem Top-Level aufgerufene Form. Implementationsabhängig enthält der Backtrace neben Formen aus dem auszuwertenden Programm auch noch vom Betriebssystem angelegte Formen. Die im Backtrace enthaltene Information ist im Fehlerfall nützlich für das Aufspüren der Fehlerursache. Deshalb verfügt ein Debugger üblicherweise über Kommandos

- um den Backtrace mit Informationen über die auftretenden Funktionen und Variablen mitsamt ihrer Bindungen ganz oder teilweise auszugeben
- um im Control-Stack in vielfältiger Weise zu navigieren
- um in einem beliebigen Frame Modifikationen und Auswertungen vorzunehmen
- um von einem beliebigen Frame aus die Auswertung wieder aufzunehmen und ggf. den Debugger zu verlassen.

In den Debugger gelangt man auch durch Auswertung gewisser Systemfunktionen, z.B. mit der Funktion ERROR. Wir haben sie bereits in Kapitel 13 kennengelernt. ERROR meldet einen sogenannten fatalen Fehler: Die Auswertung einer Funktion, die aufgrund einer Fehlermeldung mit ERROR unterbrochen wurde, kann nicht fortgesetzt werden. Sie muß neu gestartet werden.

Eine andere Möglichkeit bietet hier z.B. die Funktion CERROR, eine Modifikation von ERROR. CERROR steht für "Continue after error". Sie führt unter Abgabe einer Fehlermeldung in den Debugger; doch kann nach einer vorprogrammierten Korrektur des Fehlers zum Top-Level

zurückgekehrt und die Auswertung fortgesetzt werden. Im Zusammenhang mit der Fortsetzung kann eine weitere Meldung abgegeben werden.

CERROR hat das **Aufrufmuster:**

(CERROR <Continue-format-string>
<Error-format-string>
<Arg-1> . . . <Arg-n>) mit n ≥ 0

Auswertung:
Jedes der beiden Argumente Continue-format-string und Error-format-string wird mit den Argumenten Arg-1 bis Arg-n zur Erzeugung je einer Meldung an FORMAT übergeben. Sodann wird die Kontrolle an den Debugger übergeben. Der Error-format-string soll eine Meldung erzeugen, die den Fehler beschreibt, genauso wie bei der uns bekannten Funktion ERROR. Diese Meldung wird durch Voranstellen von ERROR: gekennzeichnet. Der Continue-format-string soll eine Meldung erzeugen, die den Effekt der Fortsetzung beschreibt, und den Benutzer nach Möglichkeit bei der Entscheidung über die Art der Fortsetzung unterstützt.

Wenn das Programm nach Auftauchen einer Fehlermeldung aufgrund von CERROR fortgesetzt wird, übergibt die CERROR-Form den Wert NIL. Anschließend wird der dem Aufruf von CERROR folgende Code ausgewertet. Dieser Code sollte den Fehler korrigieren, gegebenenfalls auch unter Übernahme eines neuen Wertes vom Benutzer, wenn eine Variable keinen geeigneten Wert hatte.

Beispiel:
Wir legen wieder die Funktion OUR-REVERSE zugrunde, behandeln aber den Fehlerfall mit CERROR. Man erkennt hier klar die unterschiedliche Bedeutung der beiden Steuerstrings. Da jeder der beiden Steuerstrings beim Zugriff auf die Folge der Argumente von vorne beginnt, wird zur Versorgung mit den "richtigen" Argumentwerten einmal ein Argument mit ~* übersprungen. Alle Eingaben des Benutzers sind fett "gedruckt".

```
-->(DEFUN OUR-REVERSE (LST)
     (DO  ((REM LST (CDR REM))
           (AUX  NIL (CONS (CAR REM) AUX)))
           ((NULL REM) AUX)
           (COND ((ATOM REM)
```

```
                (CERROR "Das Atom ~S  wird durch NIL ersetzt"
                        "~%Argument von OUR-REVERSE keine ~
                        korrekte Liste: ~S" REM LST)
                (RETURN AUX)))))
OUR-REVERSE

-->(OUR-REVERSE '(X Y Z))
(Z Y X)

-->(OUR-REVERSE '(X Y . Z))
>>ERROR: Argument von OUR-REVERSE keine korrekte Liste: (X Y . Z)

PROGN:
Original code: (PROGN (CERROR  "Das Atom ~S wird durch NIL ersetzt"
                               "~%Argument von OUR-REVERSE keine ~
                               korrekte Liste: ~S" REM LST)  (RETURN AUX))
:A     Abort to LISP Top Level
:C     Das Atom Z wird durch NIL ersetzt
-->>:C
Das Atom Z wird durch NIL ersetzt
(Y X)

-->(OUR-REVERSE 'A)
>>ERROR: Argument von OUR-REVERSE keine korrekte Liste: A

PROGN:
Original code: (PROGN (CERROR  "Das Atom ~S wird durch NIL ersetzt"
                               "~%Argument von OUR-REVERSE keine ~
                               korrekte Liste: ~S" REM LST)  (RETURN AUX))
:A     Abort to LISP Top Level
:C     Das Atom A wird durch NIL ersetzt

-->>:C                          ; Eingabe des Benutzers
Das Atom A wird durch NIL ersetzt
NIL
```

Eine andere wichtige Möglichkeit in den Debugger zu gelangen bietet die Funktion BREAK.

Sie hat das Aufrufmuster:

(BREAK <Format-String> <Arg-1> . . . <Arg-n>)

mit einem optionalen Argument Format-String und beliebig vielen weiteren Argumenten Arg-1 bis Arg-n.

Auswertung:

BREAK übergibt seine Argumente an FORMAT und veranlaßt FORMAT eine Meldung auszugeben, die zusätzlich durch Voranstellen von BREAK: gekennzeichnet ist. Sodann wird die Kontrolle an den Debugger übergeben. Dort können verschiedene Informationen, insbesondere über die aktuellen Variablenbindungen, eingeholt werden, aber keine Korrekturen vorgenommen werden.

Die Rückkehr zum Top-Level erfolgt mit einem Continue-Befehl; dabei übergibt die BREAK-Form den Wert NIL.

Break dient in dieser Form in erster Linie dazu, Haltepunkte in ein Programm einzubauen, in denen Überprüfungen vorgenommen werden können; weniger um Fehler zu melden. Es besteht auch keine Möglichkeit des Eingriffs durch eine vorprogrammierte Fehlerbehandlung wie bei CERROR. Bei Fortsetzung der Auswertung nach einem BREAK erfolgt also auch keine besondere Wiederherstellungsaktion.

Beispiel:

Wir nehmen an, unsere Funktion OUR-REVERSE sei definiert durch

```
(DEFUN OUR-REVERSE (LST)
   (DO   (  (REM LST (CDR REM))
            (AUX NIL (CONS (CAR REM) AUX)))
         (  (NULL REM) AUX)
         (COND  ((ATOM REM)
                 (BREAK  "Argument von OUR-REVERSE ~
                          keine korrekte Liste")))))
```

Wir nehmen weiter an, im Zuge der Abarbeitung einer Funktion EXAMPLE kommt es zur Auswertung von OUR-REVERSE und dabei zur Auswertung der BREAK-Form. Dies stellt sich im Protokoll wie folgt dar:

```
-->(EXAMPLE .....)          ; Aufruf von EXAMPLE
>>BREAK: Argument von OUR-REVERSE keine korrekte Liste
PROGN:
Original code: (PROGN (BREAK    "Argument von OUR-REVERSE ~
                                keine korrekte Liste"))
:C Continue on LISP Top Level
-->>LST                         ; hier wird der aktuelle Wert von LST erfragt
(A B . C)
-->>:C                          ; Eingabe des Benutzers für Rückkehr zum Top Level
NIL
-->
```

Hilfe bei der Fehlererkennung bieten auch die sogenannten TRACE- und STEP-Pakete, die wir in den folgenden Abschnitten behandeln.

22.4 Das TRACE-Paket

Mit Hilfe der speziellen Form TRACE (von to trace = eine Spur verfolgen) sind wir in der Lage, Funktionsaufrufe genau zu verfolgen. Dazu wird TRACE mit einem oder mehreren Funktionsnamen als Argumenten aufgerufen; dies bewirkt keine Auswertung der als Argumente eingegebenen Funktionen, sondern die Ausgabe von 2 Meldungen immer dann, wenn eine der angegebenen Funktionen zur Auswertung aufgerufen wird: In der ersten Meldung wird der Beginn der Auswertung der Funktion mit den zugehörigen Argumenten mitgeteilt. In der zweiten Meldung wird das Ende der Auswertung mit dem ermittelten Wert mitgeteilt. Diese Trace-Informationen sind implementationsabhängig.

Ein **Beispiel** zeigt die Trace-Information; wie sie etwa für die in Kapitel 7 definierte Funktion OUR-APPEND ausgegeben wird.

```
-->(DEFUN OUR-APPEND (X Y)
   (COND    ((NULL X) Y)
            (T (CONS (CAR X) (OUR-APPEND (CDR X) Y)))))
OUR-APPEND

-->(OUR-APPEND '(A B C) '(D E F))
(A B C D E F)

-->(TRACE OUR-APPEND)
(OUR-APPEND)

-->(OUR-APPEND '(A B C) '(D E F))
(1 Enter OUR-APPEND with arguments (A B C) (D E F))
   (2 Enter OUR-APPEND with arguments (B C) (D E F))
      (3 Enter OUR-APPEND with arguments  (C) (D E F))
         (4 Enter  OUR-APPEND  with arguments NIL (D E F))
         (4 Exit   OUR-APPEND with result (D E F))
      (3 Exit OUR-APPEND with result (C D E F))
   (2 Exit OUR-APPEND with result (B C D E F))
(1 Exit  OUR-APPEND with result (A B C D E F))
(A B C D E F)
```

Man erkennt, daß die Enter- und Exit-Meldungen für eine Funktion nicht unmittelbar aufeinanderfolgen, wenn im Rumpf der Funktion weitere Funktionen aufgerufen werden, für die ebenfalls "Tracen" angeordnet ist; im obigen Fall sind dies einfach rekursive Aufrufe der Funktion OUR-APPEND. Die Enter- und Exit-Meldungen können leicht zugeordnet werden, da das Trace-Konzept die Meldungen entsprechend der Schachtelungstiefe der Aufrufe im Druckbild einrückt und sie zugleich separat für jede Funktion entsprechend der Rekursionstiefe durchnumeriert.

Zum Trace-Konzept gehören noch folgende Werkzeuge:

- *TRACE* ohne Argumente übergibt als Wert eine Liste aller Funktionen, für die gegenwärtig Tracen vorgeschrieben ist.

- *UNTRACE* ist eine spezielle Form, deren Aufruf mit einem oder mehreren Funktionsnamen als Argumenten ein weiteres "Tracen" für die angegebenen Funktionen abstellt. So würde der Aufruf (UNTRACE OUR-APPEND) ein weiteres "Tracen" von OUR-APPEND abstellen.

- *UNTRACE* ohne Argumente bewirkt Einstellen des Tracens für alle Funktionen.

Die Funktion TRACE akzeptiert als Argumente i.a. nicht nur Funktionsnamen. In Abhängigkeit von der jeweiligen Implementation können auch kompliziertere Argumente ggf. mit Bedingungen verwendet werden, die dann zu einer vielfältigeren ggf. auch eingeschränkten Trace-Information führen.

22.5 Das STEP-Paket

Mit Hilfe des Makros STEP sind wir in der Lage, die Auswertung einer Form schrittweise zu verfolgen. Ein **Aufruf**

(STEP <Form>)

bewirkt, daß das Argument Form ausgewertet und sein Wert übergeben wird, gibt aber als Seiteneffekt dem Benutzer die Möglichkeit, die Auswertung in allen Einzelschritten zu verfolgen. Im einfachsten Fall werden alle Einzelschritte vorgeführt. Darüberhinaus können verschiedene Kommandos zur Verfügung stehen, mit denen der Benutzer gerade diejenigen Auswertungsschritte auswählt, für die er sich interessiert.

Beispiel:
Wir betrachten eine Funktion OUR-LENGTH zur Bestimmung der Länge einer Liste und rufen diese dann im Argument von STEP auf. Alle Eingaben des Benutzers sind fett gedruckt.

```
-->DEFUN OUR-LENGTH (L)
     (APPLY #'+
       (MAPCAR #' (LAMBDA (X) 1) L)))
OUR-LENGTH
```

```
-->(STEP (OUR-LENGTH '(A S D ((F)) G)))
   (OUR-LENGTH (QUOTE (A S D # G))) --> :N
      (FUNCTION OUR-LENGTH) --> :N
      #<INTERPRETED-FUNCTION (NAMED-LAMBDA OUR-LENGTH (L)
                              (BLOCK OUR-LENGTH (APPLY # #))) 4E7B97>
      (QUOTE (A S D (#)  G)) = (A S D ((F)) G)
      (BLOCK OUR-LENGTH (APPLY (FUNCTION +) (MAPCAR # 1)))  -->:PP

(BLOCK OUR-LENGTH
   (APPLY #'+
        (MAPCAR #' (LAMBDA (X) 1) L))) --> :N
      (APPLY (FUNCTION +) (MAPCAR (FUNCTION #) L))  --> :N
        (FUNCTION APPLY) --> :N
        #<COMPILED FUNCTION APPLY 267977>
        (FUNCTION +) --> :N
        #<COMPILED FUNCTION + 29591F>
        (MAPCAR (FUNCTION (LAMBDA #  1)) L) --> :N
          (FUNCTION MAPCAR) --> :N
          #<COMPILED FUNCTION MAPCAR 26D107>
          (FUNCTION (LAMBDA (X) 1)) --> :N
          #<INTERPRETED FUNCTION (LAMBDA (X) 1) 4F6CEF>
          L = (A S D ((F)) G)
          1 = 1
          1 = 1
          1 = 1
          1 = 1
          1 = 1
          (1 1 1 1 1)
       5
    5
  5
5
-->                          ; wir sind wieder auf dem Top-Level
```

Während der Abarbeitung der obigen STEP-Form wurde wiederholt das Kommando :N zur Durchführung des nächsten Schrites und einmal das Kommando :PP zum Einschalten des Pretty-Printers verwendet.

Im allgemeinen können beim "Steppen" Aktionen wie folgt angeordnet werden:

- Durchführung des nächsten Schrittes bzw. der nächsten n Schritte.
- Fortsetzung der Evaluierung unter Verzicht auf Stepping für die aktuelle Form.
- Fortsetzung der Evaluierung ohne jedes Stepping.
- Ausgabe durch den Pretty-Printer.
- Ausgabe eines Backtrace.
- Abbruch der Auswertung der STEP-Form, mit Rückkehr zum Top-Level.
- Rückkehr zum Top-Level.

Über die jeweils möglichen Aktionen kann man sich im allgemeinen mit Hilfe einer Funktion HELP informieren.

22.6 Zusammenfassung

Die Entwicklung von nachprüfbaren Qualitätskriterien für Programme ist ein schwieriges Problem. Allgemein erwartet man neben Korrektheit und Robustheit gegenüber fehlerhafter Verwendung vor allem Verständlichkeit, Testbarkeit und Wartbarkeit. So notwendig diese Eigenschaften sind, so schwer ist es, Kriterien anzugeben, mit denen das Erfülltsein dieser Eigenschaften objektiv beurteilt werden kann.

Im Hinblick auf die genannten Ziele wurde die Idee des strukturierten Programmierens entwickelt. Sie setzt voraus, daß die Programmiersprache mit geeigneten Sprachkonstrukten den modularen Aufbau unterstützt und daß diese auch entsprechend genutzt werden. Dies erfordert Ordnung und Disziplin beim Benutzer, erleichtert aber die Arbeit insgesamt und ermöglicht letztlich erst die Erstellung umfangreicher Programmsysteme.

Dabei wird der Programmierer heute durch sogenannte Programmentwicklungssysteme in allen Phasen der Programmentwicklung unterstützt, vom strukturierten Entwurf bis zur endgültigen Implementierung und Ausführung der Programme.

Besondere Bedeutung kommt dabei der Unterstützung der Fehlerbehandlung zu.

Derartige Programmiersysteme sind für keine andere Programmiersprache so weit ausgebaut worden, wie gerade für LISP. Hier ist die Unterstützung des Programmentwurfs aber auch deshalb leichter, weil LISP-Programme wie Daten verarbeitet werden können.
Durch Automatisierung verschiedener Arbeitsschritte wird der Programmentwurf insgesamt nicht nur wesentlich einfacher und schneller, sondern vor allem auch weniger fehleranfällig.

Literaturhinweise

Zur **Definition von LISP** kann man sich informieren in:

McCarthy, J. et al (1962): LISP 1.5 Programmers Manual
MIT Press, Cambridge, Mass., (2nd edition 1965).
Dieses Manual enthält den ersten LISP-Standard, der Grundlage für alle Weiterentwicklungen wurde.

McCarthy, John (1978): A Micro-Manual for LISP - Not the Whole Truth
ACM SIGPLAN Notices, Vol. 13, No. 8, 215-216.
Dieses "Micro-Manual" beschreibt den Kern der Sprachen LISP auf 2 Druckseiten!

Steele, Guy L. jr. (1984): Common LISP, The Language,
Digital Press, Burlington, Mass.
Dies ist die verbindliche Definition von COMMON LISP ! Sie ist zum Erlernen der Sprache weder konzipiert, noch geeignet, ist aber einzige und letzte Instanz zur Klärung aller Fragen.

Zur **Konzeption und Entwicklung von LISP** findet man interessante Hinweise in:

Allen, J. (1978): Anatomy of LISP, McGraw-Hill, New York.

Church, A. (1948): The Calculi of Lambda-Conversion, Princeton University Press, Princeton, New Jersey.

McCarthy, J. (1960): Recursive Functions of Symbolic Expressions and their Computation by Machine, Comm. ACM 3,4, (185-195).

McCarthy, J. (1978): History of LISP, ACM SIGPLAN Notices, Vol. 13, No. 8, 217-223.

Stoyan / Görz (1984): LISP, Eine Einführung in die Programmierung,
Springer, Berlin/Heidelberg.

Lehrbücher zum Erlernen von COMMON LISP:

Anderson, J.R.,Corbett, A.T., and Reiser, B.J. (1987): Essential LISP, Addison-Wesley, Reading, Mass.

Brooks, R.A. (1985): Programming in COMMON LISP, Wiley & Sons, New York.

Tatar, D.G. (1987): A Programmer`s Guide to COMMON LISP, Digital Press, Bedford, Mass.

Winston, P., Horn B.K.P. (1984): LISP, Addison-Wesley, Reading, Mass.

Stichwortverzeichnis

ABS 23
ACONS 154
ADJUST-ARRAY 249
:ADJUSTABLE 249
ADJUSTABLE-ARRAY-P 250, 259
A-Liste 154
AND 75
Anonyme Funktion 83, 85
APPEND 51, 170
APPLY 119, 299
APROPOS 310
Archivierungssystem 309
AREF 250
Argumente 22, 40, 83, 107-126, 197-208
  -, beliebig viele 200
  -, obligatorische 198
  -, optionale 197
  -, Schlüsselwort- 202
Argumentübergabe 89
ARRAY (Typbezeichner) 287
Array 247-260, 286, 287
ARRAY-DIMENSION 259
Array-Dimensionen 247
ARRAY-IN-BOUNDS-P 259
ARRAY-RANK 259
ARRAY-TOTAL-SIZE 259
ASSOC 156
Assoziations-Listen 154
Atom 24, 57
ATOM (Prädikat) 57, 74
ATOM (Typbezeichner) 74
at-Zeichen 189, 217
Aufrufmuster 33
Aufrufumgebung 108
Ausdruck 21, 25, 30
Ausgabe 21, 137-148, 181-195
  -, Formatierte 181-195
Auswertung, (syn.: Evaluierung) 21, 36

Backquote 216
Bäume 151
Bereitschaftszeichen ("Prompt") 21
BIGNUM 286
bildschirmorientiert (beim Editor) 307
binden 84, 107
Bindung 84, 107
BLOCK 261
BOUNDP 231
BREAK 315

Call by value 90
Call by reference 90
CAR 45, 48
CATCH 263
CDR 45, 48
CEILING 296
CERROR 312
CHAR (Funktion) 255
CHARACTER (Typbezeichner) 286
Character 286
CLOSE 138
Closure (eines Lambda-Ausdrucks) 122
COERCE 289
COMMON LISP 13
COMPLEX 286
Compilation 41
Compiler 41
COND 300
CONS 43, 48, 75
CONSP 57
CONS-Paar 49, 57
Cons-Zelle 26, 152
Control-Stack 312
COPY-ALIST 160
COPY-LIST 157
COPY-TREE 162

Datenstrukturen 285-301
Datentypen 285-301
Datum 154
Debugger 307, 311
Definitionsumgebung 108
DEFMACRO 209
DEFSTRUCT 289
DEFSTRUCT-Strukturen 289
DEFTYPE 289
DEFUN 83
DEFVAR 110
DESCRIBE 309
destruktive Funktionen 165-180
:DIRECTION 198
DO 131
DOLIST 266
DOTIMES 266
dotted list 49
dotted pair 49
DOUBLE-FLOAT 286
DRIBBLE 311
DWIM-Paket 311

Edit-Form 308
Editor 307
Eigenschaftsliste (= Property-Liste) 227
:ELEMENT-TYPE 249
ELT 254
Endrekursion 268
endrekursiv 269
:END1 257
:END2 257
Entwicklung von Programmen 303-321
EQ 61
EQL 66
EQUAL 61
ERROR 193, 312

»ERROR-OUTPUT« 138
EVAL 38, 299
Evaluierung (syn.: Auswertung) 21, 36
EXPT 23

FBOUNDP 231
Fibonacci-Zahlen 99
File 137
  -, Ende eines -s 139
  -, Laden eines -s 139
:FILL-POINTER 253
FIXNUM 286
FLOAT 286
FLOOR 296
Fluchtsymbole 145, 222
Form 25
FORMAT 184
freie Variable 90
FUNCALL 118, 299
FUNCTION 121
FUNCTIONP 232
Funktion 83-93, 286
  -, Definition einer - 83
  -, Parameter einer - 83
  -, Rumpf einer - 83
  -, Tracen einer - 316
  -, Variable einer - 90, 107-126
Funktional 121
Funktionales Argument 118
Funktionales Objekt 121
Funktionsaufruf 83
Funktionsname 83
Funktionsrumpf 83, 85

Ganze Zahlen 286
Garbage-Collection 54
GCD 23
gebundene Variable 90
GENSYM 225
GET 227
GETF 230
Globale Variable 110
Gültigkeitsbereich 108

Hanoi, Türme von - 100

IF 210
:INITIAL-CONTENTS 248
:INITIAL-ELEMENT 248
:INPUT 138
INTEGER 286
interne Struktur 25, 151
INTERSECTION 153
Interpretation 36, 41
Interpretationszyklus 36-41
Interpreter (syn.: Interpretierer) 36-41, 235-246
Interpretierer 41, 235-246
Iteration 131, 266, 268

&KEY 204
Komma 217
Kommentar 23, 24
Kontrollstrukturen 127-135, 261-271
Kopierfunktionen 157, 292

LAMBDA 85
Lambda-Ausdruck 85
Lambda-Kalkül 86
LAST 80
LENGTH 80, 254
LET 113, 214
Lexikalische Skopierung 109
LISP 13
LISP-Programm 21, 36
LISP-System 21, 303-321
LIST 51, 75
LIST* 53
LISTP 57
Liste 24, 151-164, 286
  -, Assoziations - 154
  -, Elemente einer - 25, 153
  -, Kopieren einer - 157
  -, Punkt - 49, 152
  -, Umkehrung einer - 80, 178, 254
Listen für Mengen 153
Listen, Zusammenfügen von - 51, 172
lokale Variable 90
LONG-FLOAT 286
LOOP 267

MACROEXPAND 212
MAKE- ... (Konstruktorfunktion) 291
MAKE-ARRAY 248
Makros 209
Makroexpansion 210
Makrorumpf 209
MAPC 131
MAPCAN 175
MAPCAR 129
MAPCON 175
MAPL 131
MAPLIST 129
Map-Funktionen 129
MAX 23
McCarthy 13
MEMBER 67
Mengen 153
  -, Durchschnitt von - 153
  -, Mitgliedschaft in - 153
  -, Vereinigung von - 153
modular 306
multiple values 295

MULTIPLE-VALUE-BIND 298
MULTIPLE-VALUE-CALL 298, 299
MULTIPLE-VALUE-LIST 297
MULTIPLE-VALUE-PROG1 300
MULTIPLE-VALUE-SETQ 299, 300
Mustervergleich 273-284

Namen, für Symbole 24, 221
NCONC 172
NIL 25, 57
NOT 75
NREVERSE 178, 254
NULL 71
NUMBER (Typbezeichner) 75
NUMBERP 71

obligatorische Argumente 198
OPEN 138
&OPTIONAL 198
Optionale Argumente 111, 197
OR 75
:OUTPUT analog :INPUT 138

PACKAGE 223
Package 223
PAIRLIS 154
Parameter 83, 209
Parameterliste 83, 85
Parameterlisten-Schlüsselwort 198
Pattern-Matching 273-284
Pointer 26
PPRINT 184
Prädikat 57-70
Pretty-Printer 184, 307, 309, 320
primitive Operationen 240
PRINC 146
PRINT 142
PRINT-BASE 181
PRINT-LENGTH 181
PRINT-LEVEL 181
PRIN1 146
PROG 128
PROGN 128, 300
Programmentwicklungssystem 306
Programmieren, strukturiertes - 304
Programmiersystem 306
Programmierumgebung 306
PROG1 128, 300
PROLOG 273
Prompt (Bereitschaftszeichen) 21
Property-Liste 227
Punkt-Liste 49

QUOTE 35
Quotieren 35

RASSOC 156
RATIONAL (Typbezeichner) 286
rationale Zahlen 286
READ 139
READ-BASE 183
Read-Eval-Print-Zyklus 36-41, 305
Rekursion 95-105
Rekursive Funktionen 95-105
Referenzaufruf 90
REMF 230
REMPROP 229
&REST 201
Restparameter 201
RETURN 132, 263
RETURN-FROM 261
REVERSE 80, 254
ROUND 296
RPLACA 165
RPLACD 165
Rumpf 83, 113, 131, 209

S-Ausdruck (syn.: Ausdruck) 30
Schlüssel 154
Schlüsselwort-Argumente 202
Scope 108
Seiteneffekte 31, 165-180, 218
Sequenzen 253
SETF 228, 251
SETQ 31, 300
SHORT-FLOAT 286
SINGLE-FLOAT 286
Skopierung 108
  -, dynamische 108
  -, lexikalische 108
Skopierungsregeln 108
Speicherzelle 26
Spezielle Form 38, 40
SQRT 23
STANDARD-CHAR (Typbezeichner) 287
STANDARD-INPUT 138
STANDARD-OUTPUT 138
:START1 257
:START2 257
STEP 318
Steueranweisung 185
Steuerstring 184
String 24, 38, 255
  -, Vergleich von -s 256
STRING (Funktion) 258
STRING (Typbezeichner) 287
STRING 75
STRING-CHAR (Typbezeichner) 287
STRING-EQUAL 257
STRINGP 73
STRING= 256
Ströme 137

Struktur im Speicher
(= interne Struktur) 25
Strukturen 286
strukturorientiert (beim Editor) 307
strukturiertes Programmieren 304
Supplied-Property-Parameter 200
SYMBOL 75
Symbol 24, 221-233, 286
-, Erzeugung von -en 225
-, Funktions-Zelle eines -s 226
-, Name eines -s 226
-, Package-Zelle eines -s 226
-, Property-Liste eines -s 226
-, Struktur eines -s 225
-, Wert-Zelle eines -s 226
-, Zugriff auf ein - 223
SYMBOL-FUNCTION 121, 232
SYMBOLP 73
SYMBOL-PLIST 229
SYMBOL-VALUE 231
syntaxorientiert (beim Editor) 307
Systemfunktion 31

T 24, 57
:TEST 69, 197
TERPRI 147
THROW 263
Top-Level 305, 307, 309
TRACE 316
tracen 316
TRUNCATE 296
Typbezeichner (syn.: Typ-Spezifikator) 74, 288
TYPE-OF 289
TYPEP 75, 289
Typ-Spezifikator (syn.: Typbezeichner) 74, 288

Übersetzer 41
Übersetzung 41
Umgebung 107
Umkehrung
-, von Listen 80, 178
-, von Sequenzen 254
UNION 153
UNTRACE 318
UNWIND-PROTECT 264

VALUES 295
VALUES-LIST 296
Variable 90
-, freie 90
-, gebundene 90
-, globale 110
-, lokale 90
VECTOR (Funktion) 253
VECTOR (Typbezeichner) 287
Vektoren 253
Vielfach-Werte 295

Wert **fast jede Seite**, insbes.
21, 25, 31, 36, 83-93, 107-126, 295-300
-, eines Strings 25
-, eines Symbols 25, 31, 107-126
-, einer Variablen, globaler 107-113
-, einer Variablen, lokaler 107-126
-, einer Zahl 25
Wertaufruf 90
Wertzuweisung 31

Zahl 24, 286
Zeiger 26
ZEROP 72
zirkuläre Strukturen 169
Zugriffsfunktionen 291

\ (einfaches Fluchtsymbol) 145
|...| (vielfaches Fluchtsymbol) 145
* (Funktion) 23
* (Variable) 309
** (Variable) 309
*** (Variable) 309
+ (Funktion) 23
+ (Variable) 309
++ (Variable) 309
+++ (Variable) 309
- (Funktion) 23
- (Variable) 309
/ (Funktion) 23
/= (Funktion) 65
< (Funktion) 64
<= (Funktion) 65
= (Funktion) 64
> (Funktion) 64
>= (Funktion) 65

' (Quote) 35
, (Komma) 217
` (Backquote) 216
@ (at-Zeichen) 189, 217

~% (Steueranweisung: Neue Zeile) 185
~* (Steueranweisung: Ignoriere Argument) 187
~:* (Steueranweisung: Verwende schon benutztes Argument) 187
~ (Steueranweisung: Ignoriere Neue Zeile und Leertasten) 186
~~ (Steueranweisung: Ausgabe von Tilde) 186
~[.... ~] (Steueranweisung: Fallunterscheidung) 190
~A (Steueranweisung: ASCII-Darstellung) 187
~B (Steueranweisung: Binär-Darstellung) 188
~D (Steueranweisung: Dezimal-Darstellung) 188
~E (Steueranweisung: Exponential-Darstellung von Gleitpunktzahlen) 188
~F (Steueranweisung: Floating-pointer-number) 188
~$ (Steueranweisung: Dollars-floating-point-number) 188
~G (Steueranweisung: General floating point-number) 188
~O (Steueranweisung: Oktal-Darstellung) 188
~P (Steueranweisung: engl. Plural) 189
~R (Steueranweisung: für Darstellung von ganzen Zahlen) 188
~S (Steueranweisung: S-Ausdruck) 185
~T (Steueranweisung: Tabulator) 191
~X (Steueranweisung: Hexadezimal-Darstellung) 188

Die richtige Reihe: Informatik aus dem B.I.-Wissenschaftsverlag.

Eine Reihe, die durch fundierte einführende Darstellungen zu den verschiedenen Teildisziplinen und Fragestellungen der Informatik überzeugt.
Herausgegeben von Prof. Dr. Karl Heinz Böhling, Universität Bonn, Prof. Dr. Ulrich Kulisch, Universität Karlsruhe und Prof. Dr. Hermann Maurer, Technische Universität Graz.

Albert, B.
Mikroarchitekturbeschreibung zur effizienten Programmierung anwendungsorientierter Systeme
Reihe Informatik, Band 55
399 Seiten. 1987.
Eine modulare Beschreibungssprache auf Register-Transfer Niveau. Im Mittelpunkt stehen Mikroarchitekturen, die auf Standardelementen aufbauen.

Kauffels, F.-J.
Rechnernetzwerksystemarchitekturen und Datenkommunikation
Reihe Informatik, Band 54
380 Seiten. 1987.
Logische Sicherungs- und Steuerungsverfahren bei Fernnetzen und Lokalen Netzen: Anwendungsorientiert beleuchtet der Autor die Integration dieser Netze in einheitliche Transportprotokolle.

Waldschmidt, E. H./H. K.-G. Walter
Grundzüge der Informatik I
Reihe Informatik, Band 43
338 Seiten. 1984.
Ausgehend von den verschiedenen Konzepten der künstlichen Intelligenz diskutiert der Autor anhand zahlreicher Beispiele die grundlegenden Methoden und Ansätze für die zentralen Entwurfsprobleme von KI-Systemen.

Schöning, U.
Logik für Informatiker
Reihe Informatik, Band 56
168 Seiten. 1987.
Anwendungsorientierte Einführung in Begriffe und Methoden der formalen Logik wie Resolution, Automatisches Beweisen und Logik-Programmierung.

Alber, K./W. Struckmann
Einführung in die Semantik von Programmiersprachen
Reihe Informatik, Band 59
258 Seiten. 1988.
Praxisnah beschreiben die Autoren die verschiedenen Sprechkonzepte gängiger Programmiersprachen. Der Schwerpunkt liegt auf der denotationalen Methode.

B·I

Wissenschaftsverlag
Mannheim/Wien/Zürich

Altenkrüger, D. E
Wissensdarstellung für Expertensysteme
Reihe Informatik, Band 57
314 Seiten. 1987.
Programmiersprachen, Entwicklungswerkzeuge und Maschinenarchitekturen werden im Hinblick auf Expertensysteme analysiert. Beispiele und Tabellen bieten einen anschaulichen Überblick.

Schefe, P.
Informatik – Eine konstruktive Einführung
LISP, PROLOG und andere Konzepte der Programmierung
Reihe Informatik, Band 48
406 Seiten, 2. Aufl. 1987.
Systematisch-konstruktive Einführung in die Grundbegriffe der Algorithmenformulierung: von der funktionalen bis zur von-Neumann-Programmierung.

Schefe, P.
Künstliche Intelligenz
Überblick und Grundlagen
Reihe Informatik, Band 53
227 Seiten. 1986.
Vom Algorithmusbegriff bis zur Zeichen- und Wortverarbeitung: eine fundierte und leicht verständliche Einführung in die Theorie der Syntax und Semantik von Programmiersprachen.

Waldschmidt, E. H./H. K.-G. Walter
Grundzüge der Informatik II
Reihe Informatik, Band 52
343 Seiten. 1986.
Praxisbezogen informieren die beiden Autoren über den maschinenorientierten Aspekt der Programmierung – vom Rechneraufbau bis zur Programmierung.

Zobel, R.
Diskrete Strukturen
Eine angewandte Algebra für Informatiker
Reihe Informatik, Band 49
194 Seiten. 1987.
Eine benutzerorientierte Einführung für Anwender der Algebra im Bereich der Informatik. Einige Beispiele werden als FORTRAN-77-Programme formuliert.

B·I
Wissenschaftsverlag
Mannheim/Wien/Zürich

Ausgewählte Informatik-Bücher aus dem B.I.-Wissenschaftsverlag:

Belli, F.
Einführung in die logische Programmierung mit PROLOG
277 Seiten, 1986. Kart.
Eine Einführung in die Methoden des Mensch-Maschine-Dialogs und in die Wissensdarstellung und Wissensverarbeitung mit PROLOG. Mit zahlreichen Programmbeispielen.

Grams, T.
Codierungsverfahren
162 Seiten, 1986. Kart.
Ein Buch für Computeranwender im Bereich der Bürokommunikation, Telekommunikation und Automatisierungstechnik. Im Mittelpunkt stehen die Fragen: Warum ist eine besondere Codierung erforderlich? Wie leistungsfähig sind die existierenden Verfahren?

Müller, D.
LISP
Eine elementare Einführung in die Programmierung nichtnumerischer Aufgaben
207 Seiten.
Revidierter Nachruck 1986.
Kart.

Mickel, K.-P.
Einführung in die Programmiersprache COBOL
219 Seiten, 2., verbesserte Auflage 1980. Kart.
Mit einer Einführung in die Grundzüge der EDV, in das Arbeiten mit Dateien sowie zahlreichen Programmierbeispielen.

Rohlfing, H.
PASCAL
Eine Einführung
217 Seiten. 1978. Kart.
Klare und einfach definierte Sprachkonzepte erleichtern das Erlernen des Programmierens und erlauben strukturiertes Programmieren und schrittweise Verfeinerung.

Schwald, A.
ADA
Eine Einführung
248 Seiten, 1984. Kart.
Eine kurzgefaßte Einführung in die wesentlichen Elemente der neuen Programmiersprache ADA.

B·I·
Wissenschaftsverlag
Mannheim/Wien/Zürich

Ausgewählte Informatik-Bücher aus dem B.I.-Wissenschaftsverlag:

Bohlender, G./L. Rall/Ch. Ullrich/J. Wolff v. Gudenberg
PASCAL-SC
Wirkungsvoll programmieren, kontrolliert rechnen
295 Seiten, 1986. Kart.
Dieses Buch behandelt ausführlich die neuen Sprachelemente von PASCAL-SC und illustriert die Vorteile dieser neu entwickelten Programmiersprache an zahlreichen Beispielen aus der numerischen und nichtnumerischen Datenverarbeitung.
Ein Buch, das sich unmittelbar als Vorlesungsmanuskript verwenden läßt.

Ludewig, J.
Sprachen für die Programmierung. Eine Übersicht.
196 Seiten, 1985. Kart.
Eine kurze, leicht verständliche Vorstellung der bekanntesten Programmiersprachen und ihrer Wechselwirkungen mit Programmiertechniken.

Bruderer, H. E.
Nichtnumerische Datenverarbeitung.
Dialogfähige Datenbanken und Informationsnetze - Computerschach und maschinenunterstützte Sprachübersetzung - Automation in Bibliotheken - künstliche Intelligenz und Computerkunst - Datenverarbeitung in Medizin und Chemie - Simulation von Entscheidungsprozessen - Rechtsinformatik
267 Seiten mit 60 Abb. 2. Aufl. 1986. Kart.
Ein einführender Überblick über Grundlagen, Möglichkeiten und Grenzen der nichtnumerischen Datenverarbeitung.

Gerken, W.
Datenverarbeitung im Dialog
213 Seiten, 1986. Kart.
Aufbau, Struktur und Anwendungsmöglichkeiten von Dialogsystemen. Mit vielen praxisbezogenen Beispielen.

B·I·
Wissenschaftsverlag
Mannheim/Wien/Zürich

Otto Mayer
Geboren 1939 in München.
Studium der Mathematik und Physik
an der TU München.
1968 Promotion in Mathematik
an der Universität Karlsruhe.
1971 Habilitation in Mathematik
an der Universität Karlsruhe.
Seit 1975 o. Prof. für Informatik
an der Universität Kaiserslautern.

LISP ist eine der ältesten Programmiersprachen. Sie wurde noch vor 1960 konzipiert, damals schon mit dem Ziel eine Sprache zu schaffen, die geeignet ist für Programme, die menschliche Denkvorgänge und Schlußweisen approximieren sollen. Heute ist LISP zur Standardsprache der Künstlichen Intelligenz geworden.

Der 1984 von einem internationalen Expertenteam vorgeschlagene Standard COMMON LISP ist Grundlage dieser Einführung in LISP. Sie präsentiert die grundlegenden Funktionen und Programmiertechniken in leichtverständlicher aber methodisch strenger Form. Ziel ist die Hinführung zum Programmieren und Denken in LISP.

ISBN 3-411-00638-2